

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DISSERTATION

A PATH-BASED NETWORK POLICY LANGUAGE

by

Gary N. Stone

September 2000

Dissertation Supervisors:

Bert Lundy
Geoffrey Xie

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 4

20001128 095

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2000		3. REPORT TYPE AND DATES COVERED Doctoral Dissertation	
4. TITLE AND SUBTITLE A PATH-BASED NETWORK POLICY LANGUAGE				5. FUNDING NUMBERS Order # G417	
6. AUTHOR(S) Stone, Gary N.					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA / ITO 3701 Fairfax Drive Arlington, VA 22203-1714				10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>Network policies are "traffic regulations" for the networks which make up the Internet. These are necessary for managing the flow of data, for access control to the network, and for managing the network to achieve other types of quality of service goals. However, with the myriad of different policies and networks, all with varying needs, conflicts can arise between network policies. Detecting and correcting these conflicts can be quite difficult for human administrators. Thus, there is a need for a theoretically sound method for specifying policy and for automatically detecting policy conflicts.</p> <p>This dissertation presents a path-based policy language that is more comprehensive than earlier languages for describing network policy. The Path-based Policy Language (PPL) is a formal language for constructing models of Internet service and access control. This path-based language is extensible and allows for an unambiguous representation of network policies based on both the static and dynamic attributes of today's networks. To support this language, both a compiler and policy conflict tester were developed. These tools accept network policies specified in PPL, translate them into formal logic, and using a theorem prover to test for policy conflicts. PPL allows for the efficient representation of large networks with its abbreviated path format. This path format allows multiple paths to be represented with one statement.</p>					
14. SUBJECT TERMS Policy Language, Path-Based, Network Management, Conflict Detection, Conflict Resolution				15. NUMBER OF PAGES 190	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified		19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified		20. LIMITATION OF ABSTRACT UL

SN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

Approved for public release; distribution is unlimited

A PATH-BASED NETWORK POLICY LANGUAGE

Gary N. Stone

B.S., State University of New York at Buffalo, 1987

M.S., Johns Hopkins University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

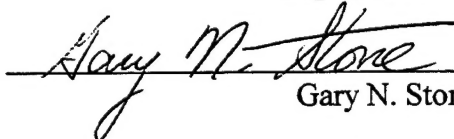
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

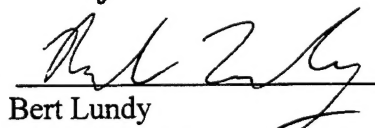
NAVAL POSTGRADUATE SCHOOL


September 2000


Author:

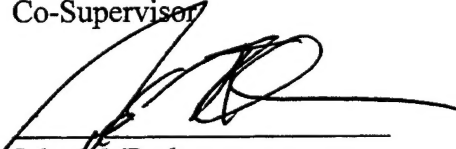

Gary N. Stone

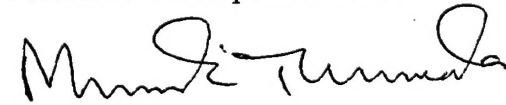
Approved by:


Bert Lundy
Professor of Computer Science
Dissertation Supervisor

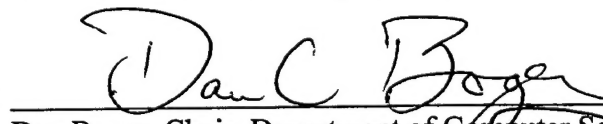

Geoffrey Xie
Professor of Computer Science
Co-Supervisor


Bret Michael
Professor of Computer Science

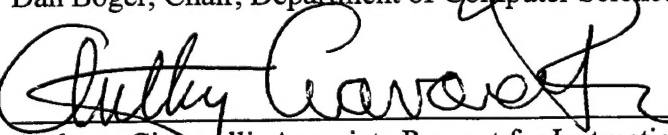

John McEachen
Professor of Electrical Engineering


Murali Tummala
Professor of Electrical Engineering

Approved by:


Dan Boger, Chair, Department of Computer Science

Approved by:


Anthony Ciavarella, Associate Provost for Instruction

ABSTRACT

Network policies are "traffic regulations" for the networks which make up the Internet. These are necessary for managing the flow of data, for access control to the network, and for managing the network to achieve other types of quality of service goals. However, with the myriad of different policies and networks, all with varying needs, conflicts can arise between network policies. Detecting and correcting these conflicts can be quite difficult for human administrators. Thus, there is a need for a theoretically sound method for specifying policy and for automatically detecting policy conflicts.

This dissertation presents a path-based policy language that is more comprehensive than earlier languages for describing network policy. The Path-based Policy Language (PPL) is a formal language for constructing models of Internet service and access control. This path-based language is extensible and allows for an unambiguous representation of network policies based on both the static and dynamic attributes of today's networks. To support this language, both a compiler and policy conflict tester were developed. These tools accept network policies specified in PPL, translate them into formal logic, and using a theorem prover to test for policy conflicts. PPL allows for the efficient representation of large networks with its abbreviated path format. This path format allows multiple paths to be represented with one statement.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
II. RELATED WORK	9
A. POLICY-BASED ROUTING PROTOCOLS	9
1. <i>Border Gateway Protocol.....</i>	9
2. <i>Inter-Domain Routing Protocol.....</i>	10
3. <i>Inter-Domain Policy Routing (IDPR).....</i>	11
B. NETWORK POLICY LANGUAGES	12
1. <i>Clark's Policy Term.....</i>	12
2. <i>Policy Framework Definition Language (PFDL).....</i>	15
3. <i>RPSL – Routing Policy Specification Language</i>	18
C. TRAFFIC FLOW LANGUAGES	20
1. <i>PAX Pattern Description Language (PDL)</i>	20
2. <i>Simple Ruleset Language (SRL)</i>	22
3. <i>Summary of Network Policy Languages.....</i>	23
D. LOGIC REPRESENTATION OF POLICIES	25
1. <i>Analyzing Consistency of Security Policies</i>	25
2. <i>On the Axiomatization of Security Policies: Some Tentative Observations About Logic Representation</i>	26
3. <i>Policy Hierarchies for Distributed Systems Management.....</i>	27
4. <i>Conflicts in Policy-based Distributed Systems Management</i>	27
5. <i>A Formal Process for Testing the Consistency of Composed Security Policies</i>	29
E. CHAPTER SUMMARY	29
III. PATH-BASED POLICY LANGUAGE (PPL).....	31
A. INTRODUCTION.....	31
B. GOALS.....	31
1. <i>Path & Non-Path Based Traffic Flows.....</i>	31
2. <i>Abstract</i>	32
3. <i>Unambiguous Policies.....</i>	32
4. <i>Detect & Resolve Conflicts</i>	32
5. <i>Dynamic Policies.....</i>	33
C. POLICIES.....	33
1. <i>Support of Integrated Services</i>	33
2. <i>Support of Differentiated Services.....</i>	37
3. <i>Support of MPLS</i>	38
4. <i>Support of Static Policies</i>	38
D. MESSAGES	39
E. PPL POLICY FORMAT.....	40
F. CHAPTER SUMMARY	42
IV. POLICY CONFLICT DETECTION AND RESOLUTION.....	43
A. INTRODUCTION.....	43
B. DETECTING CONFLICTS.....	46
1. <i>Basic Conflicts.....</i>	47
2. <i>Expanding Paths.....</i>	48
3. <i>Conditional Overlaps</i>	49
4. <i>Find Overlapping Paths using Expanded Path and Conditional Elements.....</i>	59

5. <i>Expanding Target Elements with Consideration of Action Items</i>	59
C. RESOLVING CONFLICTS	61
D. PERFORMANCE	62
E. CHAPTER SUMMARY	63
V. CASE STUDY	65
A. A 10 NODE SIMULATED NETWORK	65
B. NETWORK MODELING AND USER DEFINED DATA	66
C. NETWORK POLICIES	69
D. DETECTED CONFLICTS.....	71
E. CHAPTER SUMMARY	78
VI. CONCLUSIONS	79
A. CASE STUDY CONCLUSIONS.....	79
B. CONTRIBUTIONS.....	79
C. LIMITATIONS	80
D. FUTURE DIRECTIONS.....	81
APPENDIX A. PATH-BASED POLICY LANGUAGE (PPL)	85
1. <i>Keywords</i>	85
2. <i>Define Statements</i>	86
3. <i>Policy ID</i>	86
4. <i>User ID</i>	86
5. <i>Paths</i>	86
6. <i>Target</i>	87
7. <i>Conditions</i>	87
8. <i>Action Items</i>	87
9. <i>Legal Variables</i>	87
10. <i>Dot Quad notation</i>	87
11. <i>Reserved Symbols</i>	88
12. <i>Comments</i>	88
13. <i>Formal Grammar</i>	88
APPENDIX B. POLICY RULE INPUT FILE OF CASE STUDY	93
APPENDIX C. POLICY RULE OUTPUT FILE FOR CASE STUDY	97
APPENDIX D. PPL PARSER SOURCE CODE	103
APPENDIX E. PROLOG CONFLICT DETECTION CODE	123
LIST OF REFERENCES	171
INITIAL DISTRIBUTION LIST	175

ACKNOWLEDGEMENTS

This research was supported in part by DARPA under the Next Generation Internet Program (Order # G417), a grant from NASA Ames Research Center, and a fellowship from SPAWAR San Diego.

I. INTRODUCTION

Millions of dollars are lost, a company folds, and thousands of employees are let go, throwing a community into economic chaos. These events could result from discontinued funding from Silicon Valley investors who became aware of continued reports of poor QoS, security problems, and the inability for clients to access the company's web sites. This ingenious company was the first to integrate voice, video, and data all on the same network based on a company's policies. The company's untimely demise was the result of *conflicting network policies* that were disseminated automatically throughout the network's policy servers causing erratic network performance. This scenario, although extreme, shows the importance of being able to represent the network policy goals of a company while simultaneously verifying that those goals do not conflict with each other.

To understand how policy can play a role in managing a network, policy must be defined and applied to communication networks. The Internet Engineering Task Force (IETF) has proposed an Internet-draft of terminology for describing network policy [29] and provides many of the definitions used throughout this paper.

A policy is formally defined "as an aggregation of policy rules. Each policy rule is comprised of a set of conditions and a corresponding set of actions. The conditions define when the policy rule is applicable. Once a policy rule is so activated, one or more actions contained by that policy rule may then be executed. These actions are associated with either meeting or not meeting the set of conditions specified in the policy rule" [29]. In other words, a policy specifies what action(s) must be taken when a set of associated conditions are met.

A simple view of policy in regards to networks is that policy constrains communication. Specifically, network policy defines the relationship between clients using network resources and those network elements that provide those resources. A *client* in this case refers to users as well as applications and services.

Network policy allows administrators to manage network elements to provide service to a set of clients. If every system were permitted to communicate with all other systems without restriction, then there would be no need for network policies. Increasingly, networks that once only supported best-effort traffic are now integrating voice and data as well. Without a means for network managers to control the use of the network, mission-critical applications and general

network performance is going to suffer and there will be little hope of supporting future real-time applications.

Network policies are grouped into three general areas:

1. how the policy is used
2. how the policy is triggered
3. what level the policy is applied

A *usage policy* describes *what* services will be used to maintain the current state of the network or to transition to a new state. Services which may be available in the network are differentiated service classes, virtual private networks, encryption capability, etc. A usage policy also describes *how* those services will be used. For example the ability to differentiate the handling of separate flows of traffic based on the service class they reside in, or which virtual channel they belong to, describes how a service is used.

Policies can be *triggered* in two ways, either statically or dynamically. “Static policies apply a fixed set of actions in a pre-determined way according to a set of pre-defined parameters that determine how the policy is used”[29]. Examples of static policies are:

- transit traffic is not permitted during normal working hours
- Internet radio is only permitted after 4:00 PM
- for security reasons certain network addresses are denied access to network resources

Dynamic policies are only enforced when needed, and are based on changing conditions of the network such as congestion, packet loss, or the loss of a network router. To support the dynamic and sometimes unexpected nature of the network, actions can be triggered when an event causes a policy condition to be met. Examples of dynamic policies are:

- when the network gets congested, streaming video traffic is disallowed;
- when gold class user is utilizing the network, lower best-effort traffic to only 25% of link capacity.

The *level of the policy* can also be applied as a category. These policies are differentiated by their granularity, such as the application level, user level, class level, or service level. For example a mission critical application may be given priority over all other network traffic, or all users in the silver class (differentiated service) have priority over the bronze class but must succumb to the gold class.

In this thesis, a network policy language based on path is introduced. *Path-based policy* is defined to be a policy where all attributes associated with the policy, which include the service type of the traffic, conditions used to trigger the policy, and the actions executed when the policy is triggered, are all bound to a predefined path. Using path as the fundamental building block of a policy statement provides great control and flexibility. The ability to specify an explicit path, which represents each node from source to destination, enables us to create virtual channels where resources are reserved to support real time applications. These paths can either be specified by a user or by a network administrator. If a path had the restriction of *always* including each node in the path, then the number of unique paths needed to support a network could soon become overwhelming. This is why a path may include wild card characters, and thus adds great flexibility to the way policies are specified. The use of a wild card character allows for path aggregation which greatly reduces the number of paths that have to be specified, and at the extreme one path statement can specify all possible paths under an administrator's control.

Policy-based networking - the ability to control a networking environment by specifying and enforcing policies - is gaining increased interest among the network community. Policy-based networking helps manage user and applications priority, quality of service and security rights, based on management policies. Because of an increasing industry trend to deploy business applications over the network and the convergence of voice, video and data applications on the same network, major network vendors such as Cisco, Nortel, and Lucent Technologies are developing products to support network management. These products allow network managers to create and implement policies that can prioritize the use of network resources by different network applications so that bandwidth will be guaranteed to the most business-critical applications during times of network congestion. For example, a company which offers IP telephony - which has strict timing requirements - must not permit a large data file transfer to interfere. Network management also provides the ability to restrict the use of network segments by denying access of unwanted and perhaps malicious traffic. The ability to create and enforce network policies adds intelligence to a network that was previously based only on best-effort packet traffic. Rather than adding more bandwidth, which is expensive and time consuming, to solve existing network congestion, companies can use network policies to allow for important applications and user groups to receive network priority over secondary network users.

Many aspects of policy-based networking are being addressed such as policy storage structures, policy servers, and protocols to deliver translated policies to enforcement points. One aspect of policy-based networking that does *not* seem to be receiving much attention is the verification of policies that are going to be applied to the network. Consistent enforcement of network policies, often specified by different people at different times, is impossible if those policies conflict with each other. Thus, a method is needed to detect and deal with conflicting policies before they are distributed throughout the network to the policy enforcement points.

IETF Policy Framework Core Information Model

With the emergence of service models such as Differentiated Services (DS) [25,37], Integrated Services (IS) [25,37], and Multiprotocol Label Switching (MPLS) [36,37], the IETF has published a working draft for terminology to describe network policies and services [29]. This draft attempts to develop a scalable framework for policy administration and distribution of network policies across multiple devices and multiple vendors. A key to this framework is a common language to represent and provide a consistent implementation of policy.

An underlying assumption of this draft is that policies are stored in a centralized repository. The policy repository is one of three important entities of the model. The other entities are the Policy Enforcement Points (PEP) and Policy Decision Point (PDP).

The PEP is a component of a network node (e.g., a router, switch, or hub) where the policy decisions are actually enforced. When the PEP requires a policy decision about a new flow of traffic, or authentication for example, the PEP will send a request to a PDP.

The PDP is the entity in the network where policy decisions are made. This PDP, which may reside on a remote server, will make policy decisions using information retrieved from policy repositories.

Communication is needed to and from the policy repository as well as between the PDP and the PEP. In many proposals the policy repository is a directory and therefore the appropriate access protocol would be the Lightweight Directory Access Protocol (LDAP). Examples of a policy protocol, which is used to request and reply to policy decisions, could be the Common Open Policy Service protocol (COPS) [3] and the Simple Network Management Protocol (SNMP) [5].

Figure 1 shows the components of a generic policy-based network management architecture. Although each of the components are displayed separately in the figure, they do not have to necessarily be implemented this way.

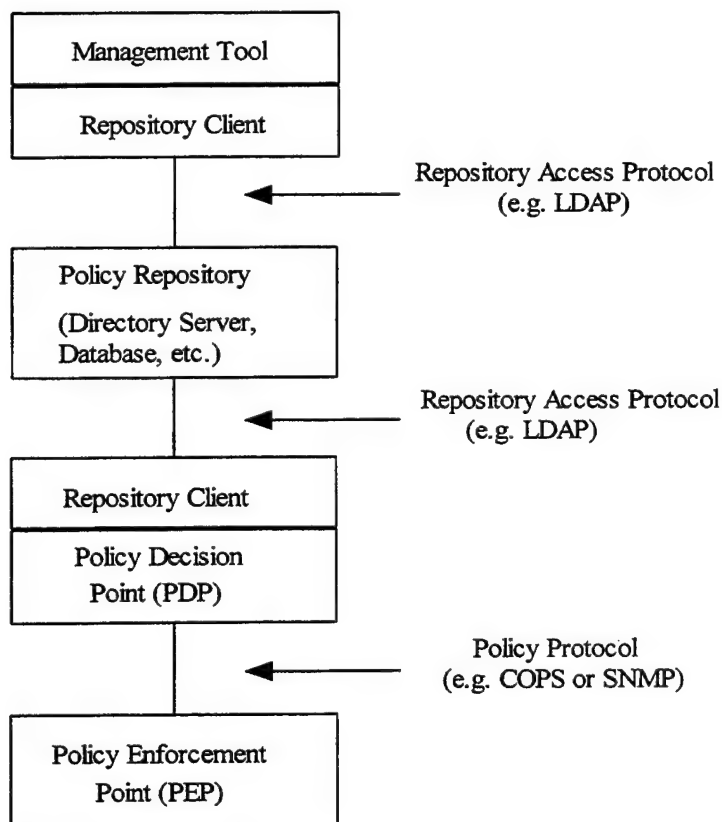


Figure 1. Generic Policy Based Architecture. After [29]

Since the PEPs can potentially be from multiple vendors, a common policy language is needed to support the dissemination of policy information to these devices. In the Policy Framework Core Information Model [32], policy is defined as an aggregation of policy rules. Each of these policy rules is composed of a set of conditions and a set of actions to perform if the conditions are met. The general form of these conditional statements is shown below.

IF <condition 1> **AND** <condition 2> ... **AND** <condition N>
THEN <action 1> ... **AND** <action N>

The policy representation includes a means to prioritize and order both the conditional statements as well as the policy actions. This is crucial when multiple policies exist and these policies conflict. A conflict occurs when the conditions of at least two policies are simultaneously satisfied, but the actions of at least one of the policies can not be simultaneously executed. For example, a router may have two access control rules where their conditions are simultaneously satisfied, but one contains that action deny, the other permit. For example:

```
access-list 1 permit 131.1.30.0 0.0.0.255  
access-list 1 deny 131.1.0.0 0.0.255.255
```

The first permits traffic with IP addresses beginning with 131.1.30 to pass. The second rule conflicts with the previous one by denying traffic with any IP address beginning with 131.1. The first rule in an access list that satisfies the conditional requirement is executed. This procedure resolves conflicts but puts the onus on the operator to enter the rules in the correct order.

It was hypothesized that it is possible to detect conflicting network policies when those policies are specified using an unambiguous language. To support this hypothesis, a language was designed that can represent network policies in an abstract and unambiguous way. Using formal methods a compiler was implemented that can detect conflicts between multiple network policies. It was also hypothesized that it is possible to resolve some policy conflicts automatically. This resolution is done with a limited scope, but the concept can be expanded to support other conflict resolution schemes.

It is envisioned that the Path-based Policy Language (PPL) compiler and conflict tester will be utilized in a network such as depicted in Figure 2. A secure user interface will be needed to control access to the PPL configuration file, which contains network connectivity information as well as the policies used to regulate the network. A cycle should develop as the network policies are created and modified until when composed together, they no longer generate conflicts. Once a conflict free set of policies has been specified, they will be distributed to a policy server where they can be used to regulate the network. It is expected that network devices will provide information about the current state of the network, such as delay, loss_rate, etc., and this information along with the specified policies will be used to response to requests made for

network resources. A protocol such as the Simple Network Management Protocol (SMTP) could be used to provide status about the current state of network, which then could be used in the decision making process of allocating resources.

Existing protocols such as the Resource Reservation Protocol (RSVP), could be used to make requests to the policy server in order to reserve network resources. The policy server would then apply network policies, as well as the current state of the network to the decision making process. New initiatives such as the Server and Agent based Active network Management (SAAM) [38] project, could also utilize this technology. SAAM makes efficient use of network resources in the support Integrated Services. The ability to extract the complex decision making process from overtaxed routers, and place it into a small number of dedicated servers, fits very well into the SAAM architecture.

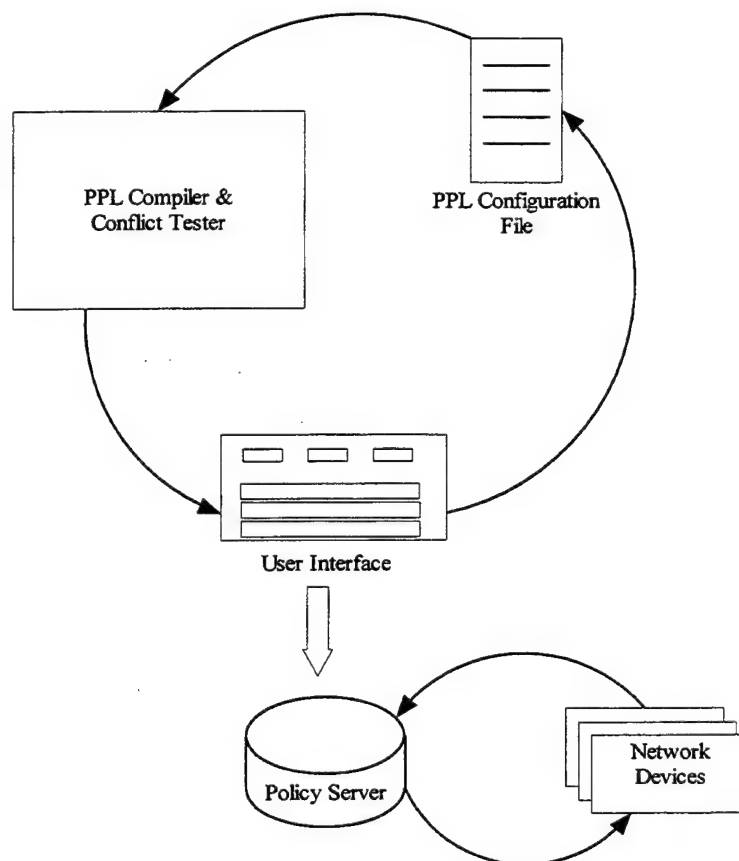


Figure 2. Network Configuration Utilizing PPL

The rest of this thesis is organized as follows:

Chapter II reviews work related to the representation of network policies, as well as the use of formal logic to detect conflicting security and management policies. These representations of network policy are based on different levels of abstraction. These levels are policy-based routing protocols, high level representations without low-level details, and languages based on the representation of patterns that can be used in the selection of network traffic. A lot of early work on the use of policies in networks occurred in the context of these network policy representations. The formal logic section reviews previous efforts of using formal logic to determine consistency between policies.

Chapter III introduces the Path-based Policy Language (PPL). The goal was to represent network policies at an abstract level in order to support heterogeneous networks, while also providing the translation of those policies into formal logic. Having policies represented in logic will provide the ability for a compiler, which uses the Prolog (PROgramming in LOGic) language, to detect conflicting network policies.

Chapter IV presents a formal process for testing the consistency of multiple network policies applied to the same network segment.

Chapter V represents the formal process of detecting, and the possible resolution of conflicting network policies applied to multiple case studies.

Chapter VI presents the conclusions, contributions, limitations, and future direction of network policy representation and conflict detection using PPL.

II. RELATED WORK

A. POLICY-BASED ROUTING PROTOCOLS

In this section three policy-based routing protocols are reviewed. These routing protocols provided a lot of early work on the use of policies in networks. All of these protocols enable policies to be enforced based on the elements of an explicit path through the network.

1. Border Gateway Protocol

Lougheed and Rekhter define an inter-autonomous routing protocol, Border Gateway Protocol (BGP) [13, 14, 15], where routers share reachability information by passing Autonomous System ¹(AS) information between neighbors. This exchange of routing information contains full AS paths that the traffic will transit to reach a distant network. Path information is not only useful in removing loops in the network, but also allows policy decisions to be made at the AS level. Policy enforcement is not part of the protocol itself but instead is manually configured at each BGP router.

Policy decisions made by BGP [16, 9] are based on configuration information manually configured into each router by an AS administrator. The enforcement of policies is accomplished in two ways. The first is by specifying the procedure by the AS router itself to select the best paths, and the second is by controlling the redistribution of routing information to neighboring ASs.

Policy decisions can be based on various preferences and constraints. Since the complete AS path is advertised to neighboring routers, particular paths can be rejected based on an AS that is contained in the path. The reasons a particular path are rejected vary. For example a particular AS whose control is under that of a major competitor may want to be avoided, causing one or more paths that include this AS to be eliminated from consideration. Performance information can also be used to eliminate paths from consideration. If an AS has access to metrics related to

¹ Autonomous System (AS), Administrative Region (AR) are a set of routers under a single technical administration, using one or more interior gateway protocols to route packets within the AS, and using an exterior gateway protocol to route packets to other ASs[n3].

performance such as link speed, delay, or capacity, then these measurements can be used to rate multiple paths for selection.

BGP allowing an AS to control redistribution of routing information is the means by which an AS can enforce policies on others. For example, if an AS does not want to be used for transit traffic, then it does so by not advertising routes to networks other than those directly connected to it.

Fundamentally BGP is a distance vector protocol, but instead of maintaining just the cost to each destination, BGP keeps track of the exact path used. As mentioned earlier policies are not part of the BGP protocol itself and therefore each AS may have its own means for evaluating paths. Each router contains a module for examining paths to a given destination and scores them. This scoring mechanism, which may include local policy information, is then used to choose the best path to a destination.

BGP routers can only advertise paths that itself uses. This prevents an AS from sending datagrams to a distant network using one path, but advertising an alternative path for others to use. This "hop-by-hop" routing paradigm which is generally used by the current Internet prevents the support of source routing.

2. Inter-Domain Routing Protocol

Kunzinger and Thomas describe the Inter-Domain Routing Protocol (IDRP) [10, 34], which is the International Standards Organization's (ISO) protocol for routing between Autonomous Systems. Just as in BGP, IDRP supports policy-based routing, but is not concerned with the implementation details of those policies. Policy-based routing can restrict access, and therefore enforce policy, by controlling the distribution of routing information to neighboring routers. This selective distribution of information can enable the AS to deny all transit traffic, or may deny access to only certain network paths.

The IDRP router accepts router information from neighboring routers, which express their views of the network, and uses this gathered information to construct its own view of the network. The IDRP router at this point can use local policy information to select or reject routes accordingly. The IDRP router advertises its view of the network with internal gateway protocols such as Open Shortest Path First (OSPF) or Routing Information Protocol (RIP) so that all routers within the AS have a consistent view of the network.

Just as autonomous systems were used to refer to an entire set of IP networks, IDRP supports a concept called routing confederations. A routing confederation is a grouping of autonomous systems to make managing the Internet more manageable. As the Internet has grown, the number of autonomous systems has also grown making its management less efficient. These routing confederations are quite flexible in that they can be subsets of each other, and can even overlap each other.

IDRP uses path vector routing to propagate routing information. Path vector routing, like BGP, explicitly lists the entire path to each destination. This concept can alleviate network loops as well as enforce policy constraints based on the autonomous systems or confederations that comprise the path.

Another feature supported in IDRP is the ability to reduce the number of path vectors by using route aggregation. Route aggregation lets an IDRP router combine multiple IP address prefixes, destinations, to create a single advertisement for them all. This feature greatly reduces the number of individual destinations a router must support as well as reducing the amount of data that has to be sent during the advertising phase.

3. Inter-Domain Policy Routing (IDPR)

Steenstrup presents a set of protocols [28] and an architecture in [27] for Inter-Domain Policy Routing (IDPR). IDPR is a routing protocol that provides policy routing among Administrative Domains (ADs)². The primary objective of IDPR is to provide traffic with routes that satisfy the users' service requirements while respecting the service providers' service restrictions [26]. *Source policies* represent the users' requirements and can consist of parameters, such as throughput, acceptable delay, cost of session, domains to avoid, etc. Service providers specify *transit policies* which specify offered services and the conditions of their use.

During route generation and selection, routes are filtered out which are not consistent with both the source and transit policies. Route generation is inherently complex and the most computationally intensive part of IDPR. The general policy route generation problem involves a

² Administrative domain (AD) refers to any collection of contiguous networks, gateways, links, and hosts governed by a single administrative authority who selects the intra-domain routing procedures and addressing schemes, specifies service restrictions for transit traffic, and defines service requirements for locally generated traffic.

combination of service constraints. For example, finding a route that generates minimum-delay and least-cost. Trying to calculate such a route is an NP-complete problem.

To transport data along a selected route, a hop-by-hop or source specific method can be used. With hop-by-hop forwarding, each router makes an independent forwarding decision based on its forwarding information database. If all the routers have consistent information then the result is the same as source specific. With source specific, the source domain dictates the data message forwarding decisions to the routing entities in each intermediate domain, which then forward data messages according to the source specification.

To reduce the size of the link-state database, IDRP supports the ability to group ADs into super domains. The existence of super domains imposes a domain hierarchy within the network. With a hierarchical approach only domain level information is needed to construct routes. This greatly reduces the information needed to be maintained by a route server. The size of the database will now depend on the number of domains and the policies associated with each.

A variant of Clark's policy term, Section B.1, was chosen to represent policies in [28]. This variant allows for policies to be associated with a set of network elements that represents a path. A policy based on path is a great asset to policy-based routing protocols.

B. NETWORK POLICY LANGUAGES

In this section all the major policy languages are discussed. These languages are used to represent varying types of network policies such as routing, access, and QoS.

1. Clark's Policy Term

Seeing the importance of using network resources differently and more efficiently, Clark proposed a template to represent network policies [7]. This template, called a *Policy Term*, was designed to enable a wide range of network policies to be represented. The work is based on the fundamental assumption that Internet resources are grouped into Administrative Regions (ARs). ARs resources included such items as networks, links, routers, and gateways. The format of a Policy Term is shown in Figure 3.

The first two "elements" of the Policy Term represent the source and destination points respectively. Each of these two points consist of three parts which provide for a wide range of

granularity while specifying the end points. To show the granularity available with this schema, here are some examples of source and destination points that could be represented with the diagram in Figure 4. These source and destination points use the special characters "*" and "-". The "*" represents the wild-card

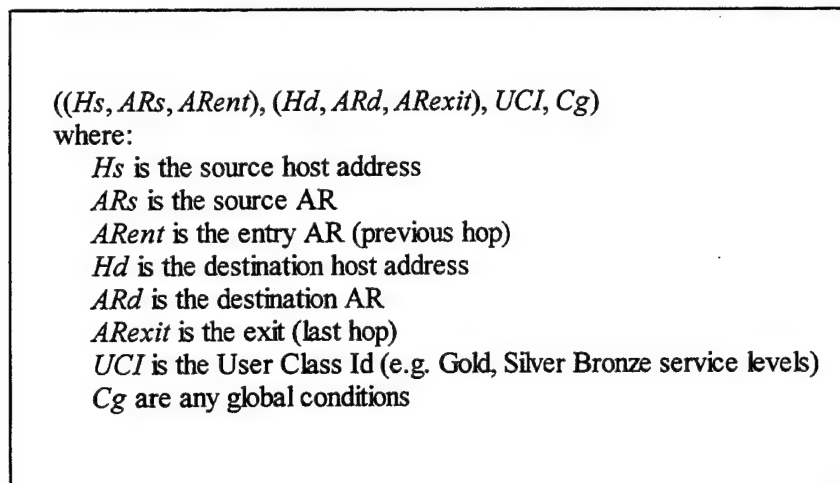


Figure 3. Policy Term. After [7]

match and the "-" is used to make sure the AR entry or AR exit fields match the source AR or destination AR respectively. These examples could be applied to AR 2.

$(*, *, *) (*, *, *)$

No restrictions, allow all traffic flows to traverse without restriction.

$(*, 36, -) (*, 12, *)$

Allow all hosts directly attached to AR 36 to pass if their destination goes through AR 12 (e.g., host 131.120.1.13 may communicate with 216.34.20.1)

$(131.120.1.13, 36, -) (216.32.74.53, 2, -)$

The host with IP address 131.120.1.13 in AR 36 may communicate with the host with IP address 216.32.74.53 in the AR 2.

As the reader can see, the end points can be as explicit as specifying a host or generic enough to allow all Internet traffic. Although the use of these first two elements provides for a

flexible way to permit traffic flow, it can become cumbersome at times. If, for example, the reader wanted to allow only traffic from universities to flow across an AR, it could be accomplished by creating many policy terms, one for each university. This list of policy terms could become quite large, so the third “element” of the policy term, UCI, can be used to make the implementation of this policy more manageable. A policy term that would only allow university traffic to flow across an AR could be represented like this:

((*,*,*), (*,*,*), University, *)

The end points are such that if the UCI element was not used, then all traffic would flow across the AR. Using the UCI as a filter, only traffic marked with a **University** tag would be permitted to pass.

The last element field of the policy term, Cg, is used for global conditions. Examples of information that might be held in this field are, “unauthenticated UCI”, “no-per-packet charge”, and “limited to n% of available bandwidth”.

((*,*,*),(*,*,*), University, {unauthenticated UCI})

This would allow only traffic marked as **University** to flow through the AR. There is no need to verify that the packet traffic was really from a university host.

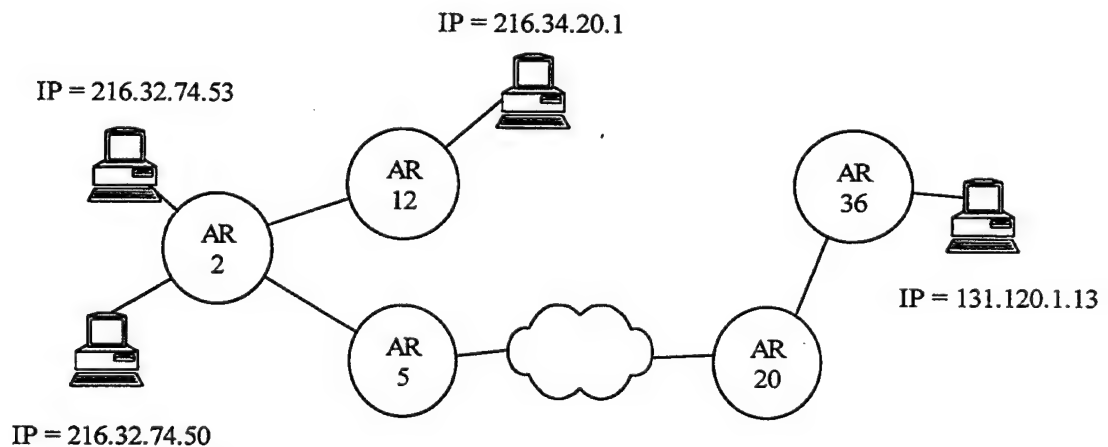


Figure 4. Sample Network Diagram

Although this was a good start for an abstract network policy representation, which is needed for heterogeneous environments, it has limitations. The first limitation is that there is no ability to represent explicit paths formed by a sequence of ARs as part of the term. Only single ARs and the wild card character "*" are allowed. Without this capability, support of Integrated Services requires several network policies distributed throughout the network to be combined for verification of a path. Consequently, there is no ability to exclude a set of ARs from a term to which a general policy is applied. Lastly, it may be desirable to represent a policy that is directional, so that a connection can be opened in one direction that has different conditions than the reverse flow. As defined in the language policy terms are bi-directional.

2. Policy Framework Definition Language (PFDL)

Strassner and Schliemier define the language *PFDL* [31] that provides a mapping of network service requirements from a business specification to a vendor- and device- independent format. The benefits of such a language is that network policy can exist in a heterogeneous environment of devices that support policy enforcement.

With the development of standards to provide QoS, like integrated services with RSVP³ and differentiated services, the IETF working group on Policy Management has proposed this language. The belief is that without a means for representing, administrating, and distributing consistent policy information, these QoS standards that classify and give preferential treatment to certain types of traffic flows will not see wide-scale deployment.

In this first release of the draft, the grammar was only available in Backus-Naur Form (BNF) and no explicit examples were presented. Attributes the authors believe should be supported by the language are discussed. As with many of these efforts to represent policy, the authors believe that having a language that will support multiple network devices and vendors is the key to successful policy deployment.

The design of PFDL is based on the Common Information Model (CIM) [32] being designed by the Distributed Management Task Force (DMTF). This model defines a hierarchy of object classes that can be used to represent policy information.

³ Resource Reservation Protocol defines how applications can place reservations, and how they can relinquish those resources once their need ends.

The class and relationship hierarchy of the CIM model are used to help define the structure of the PFDL grammar, see Figure 5. The basic premise is that a policy is an aggregation⁴ of policy rules. A policy rule defines a sequence of actions to be initiated when a corresponding set of conditions is satisfied. Five classes defined to support the CIM are the *ComplexPolicy* class, *SimplePolicy* class, *PolicyRule* class, *PolicyCondition* class, and the *PolicyAction* class. Their relationship to each other is shown in Figure 5. A *PolicyRule* contains a set of *PolicyConditions* and a set of *PolicyActions*. When the set of *PolicyConditions* are met, the set of *PolicyActions* will be executed.

A *PolicyConditionStatement* is composed of a category and value pair. These two components are specific to a particular knowledge domain, whether the domain be QoS, security, or any other domain. Providing conditions and actions for a given knowledge domain accommodates the interoperability requirement for the language. It will provide the means for multiple vendors to supply components to a general policy architecture.

A *PolicyAction* is a class in the PFDL model that consists of an action or a list of actions that will be executed when the conditions associated with a policy are evaluated to true. These actions can either be executed in a specific order, or any order which is the default. Along with the ordering of policy actions, the ability exists for the conditional execution of one or more actions based on the results of previous actions. The reader can see from Figure 5. that the hierarchy of the *PolicyActions* class is similar to the *PolicyConditions* class.

With possibly hundreds, perhaps thousands of policies to be supported in a network, the ability to detect conflicting policies is crucial. The authors of PFDL are aware of the need to both detect as well as support facilities to resolve conflicts. This proposal groups policy conflicts into two different categories, intra-policy and inter-policy conflicts.

Intra-policy conflicts are caused when the conditions of at least two policies are simultaneously satisfied, but the execution of the actions of these policies cannot be executed at the same time. Inter-policy conflicts are described as two or more policies that, when applied to the network, result in conflicting configuration commands to be specified for one or more network devices. In this case, the conflict exists when the policy is applied to a specific network or device(s). An example given in the proposal is when two policies are executed such that the

⁴ An aggregation is a string form of an association. An aggregation is usually used to represent a "whole-part" relationship.

number of queues in one network device is such that it does not match the number of queues allocated in a second device supporting the same traffic flow.

Once conflicting policies are detected, they may be resolved in several different ways. The most obvious would be to modify the conditions or actions of the policies to remove the conflict. If this cannot be accomplished and the conflicting policies must exist in the system, there are three different ways to resolve them:

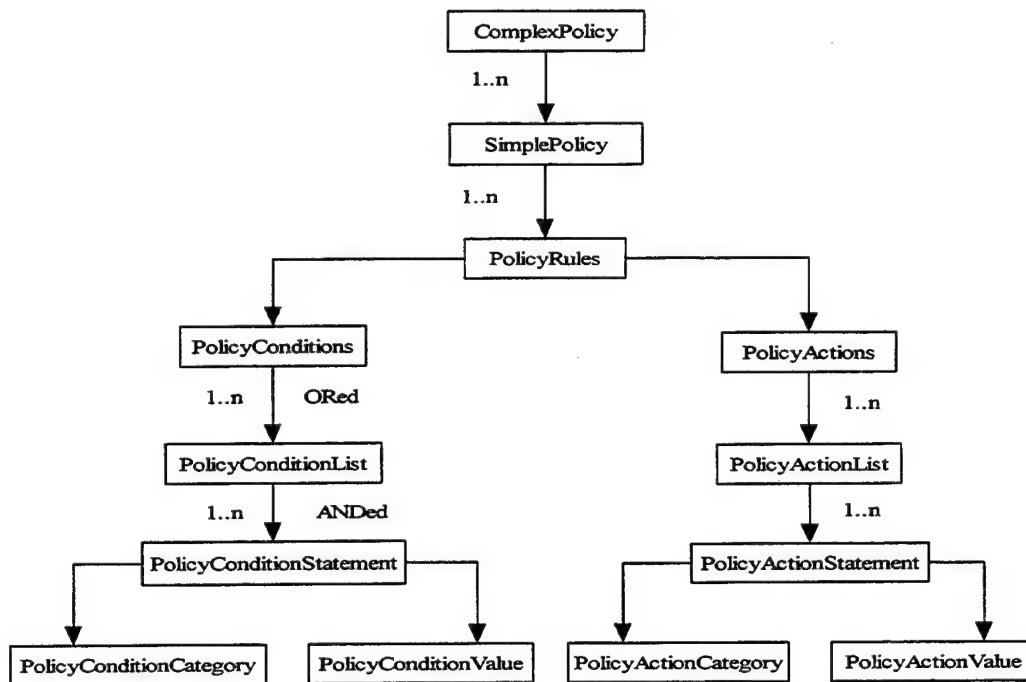


Figure 5. PFDL Hierarchy. After [32]

1. Resolve the conflict by only executing the first policy in the conflicting set.
2. Use a priority scheme where only the highest priority policy in a conflicting situation will be executed.
3. Use some type of metadata to determine which rule should be applied. The difference between this and straight priority is that priority is inherently linear, whereas metadata enables non-linear solutions, such as branching, to be used.

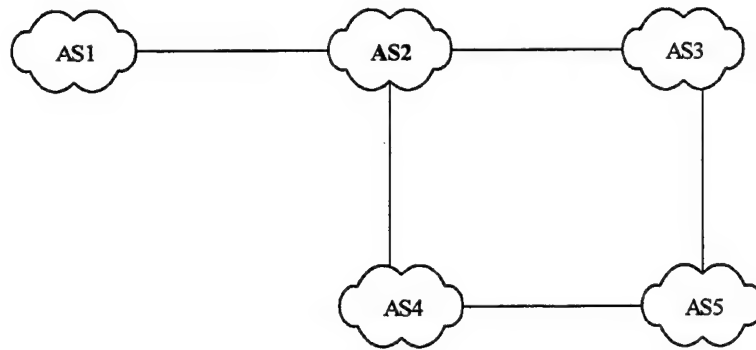
PFDL does not support path-based policies. A Path-based capability aids in initiatives such as Integrated Services [25, 37] and SAAM (Server and Agent based Active network Management) [38]. PFDL is a nice high-level framework, but lots of details need to be filled in.

3. RPSL – Routing Policy Specification Language

One of the activities of the Routing Policy System working group of the IETF is to develop a language for describing routing policy constraints. Alaettinoglu, Meyer et. al. provide a reference for the language [1] and a guide on how to use the language [19]. RPSL is a replacement for RIPE-81, the first language deployed in the Internet for specifying routing policies, and is the current Internet policy specification language. Specifying policies in RPSL allows a network operator to specify routing policies in the Internet Routing Registry (IRR), so that policies and announcements can be checked for consistency. The IRR stores the object-oriented policies of authorized organizations so that they can be queried by others using the whois⁵ service. Each object which contributes to a policy stores pieces of information regarding the policy. Each object used to represent the policies contains attributes referred to as keys, that can either be mandatory or optional. RPSL is designed so that router configurations can be generated from the policies described with the language.

Figure 6 is an example from [19] that represents a common but perhaps simple policy. The *aut-num* represents the Autonomous System number, in this case AS2 represents autonomous system 2. The *as-name* and *descr* attributes are the Autonomous System's name and description, respectively. The most important attributes of this aut-num are the import and export policies. The import clause specifies the import policies, while the export clause specifies export policies.

⁵ WHOIS is used to look up records in a Whois database. Each record has a "handle", a unique identifier assigned to it by the Network Information Center (NIC). Each whois record will also have a name, a record type, and various other fields of information, all depending on the type of whois record



```

aut-num:AS2
as-name:CAT-NET
descr:      Catatonic State University
import:     from AS1 accept ANY
import:     from AS3 accept <^AS3+$>
export:     to AS3 announce ANY
export:     to AS1 announce AS2 AS3
admin-c:    AO36-RIPE
tech-c:     CO19-RIPE
mnt-by:     OPS4-RIPE
changed:    orange@ripe.net
source:     RIPE
  
```

Figure 6. RPSL Diagram and Policy Example. From [19]

In this example, the import policy of “from AS1 accept ANY” indicates that AS2 will accept any announcements that AS1 sends. The second import policy states that AS2 only accepts announcements from AS3 which originated in AS3 and have paths composed of only AS3’s.

The export policy of “to AS3 announce ANY” indicates that any route that AS2 has in its routing table will be passed on to AS3. The second export will allow the announcements of all routes from AS2 or routes learned from AS3 to be sent to AS1.

The *admin-c* (administrative), *tech-c* (technical), *mnt-by* (maintained by), and *changed* (last changed by), are attributes that contain contact information. The values assigned to these

attributes are handles that uniquely identify the person responsible for the attribute. The *source* entry indicates that this object belongs to the RIPE⁶ registry.

RPSL represents routing policies well, but was not intended for supporting policies regarding QoS or general access control mechanisms.

C. TRAFFIC FLOW LANGUAGES

This section discusses languages that are used in the selection of network traffic in the conditional section of a policy. At a lower level than the languages in Section 2, these languages can be used for pattern matching in network devices.

1. PAX Pattern Description Language (PDL)

Nossik, Welfeld, and Richardson describe PAX [23], a special purpose language used for defining pattern matching criteria in policy-based networking devices. PAX was intended primarily for data communications networks, but is also generic enough to be used for any kind of pattern recognition.

The language itself was designed to be much like that of the C programming language. Viewing the code of a PAX program, the reader will see features similar to C such as comments, preprocessing directives, source file inclusion, conditional compilation, import and export statements, and the use of defines and macros.

The basic concept in PAX is the pattern, with simple patterns being combined to form more complex patterns. The use of field concatenation, field combination, and the ability to name patterns leads to a flexible and powerful language for describing patterns in data communication.

Examples from [23] will provide the reader with a quick idea of the syntax and features of the language. Two built-in fields used in the following examples are BIT and UINT and are used to create the simplest of patterns.

⁶ The RIPE Network Coordination Centre acts as the Regional Internet Registry (RIR) for Europe and surrounding areas

BIT 16 - matches any 16 bits in the input

UINT 4 - matches any 4 bits and value of those bits are converted to an unsigned numeric field

Figure 7 illustrates a pattern to match IP version 4 headers of TCP/IP non-fragmented packets without IP options. This figure shows how simple patterns can be concatenated together to form more complex patterns. This pattern matches the input only when the 4 bit “version” element is equal to decimal 4, the next element “ihl” equals the decimal value 5, and the subsequent simple patterns are all successfully matched.

```
{
    version UINT 4 == 4;           /* IP version 4 packet */
    ihl UINT 4 == 5;               /* length == 5 : no options */
    typeOfService UINT 8;
    totalLength UINT 16;
    identification UINT 16;
    flagReserved BIT 1 == 0;
    flagDontFragment BIT 1;
    flagMoreFragments BIT 1 == 0; /* last fragments only */
    fragmentOffset UINT 13 == 0;   /* first fragments only */
    timeToLive UINT 8;
    protocol BIT 8 == 6;           /* Next protocol TCP */
    headerChecksum BIT 16;
    sourceAddress BIT 32;
    destinationAddress BIT 32;
}
```

Figure 7. PAX Pattern to Match IPv4 Header. From [23]

Figure 8 represents two more features of the PAX language. The first being the ability to name the pattern for inclusion in other more complex patterns, in the case the name being *IEEE_802_2_LLC*. The second feature illustrated here is the use of a conditional field. Conditional fields are used to describe patterns with varying layouts depending on previous fields. In this case when the *Control1* field is equal to *0b11*, the next 6 bits are used to create a field called *ShortControl*. When the value of *Control1* is not equal to *0b11*, then the next 14 bits are used to create a field called *LongControl*.

```

PATTERN IEEE_802_2_LLC {
    DSAP BIT 8 <> 0xFF; /* Destination SAP not broadcast */
    SSAP BIT 8 <> 0xFF; /* Source SAP not broadcast */
    Control1 BIT 2;
    LongControl BIT 14 WHEN Control1 <> 0b11;
    ShortControl BIT 6 WHEN Control1 == 0b11;
}

```

Figure 8. PAX Pattern with Conditional Field. From [23]

2. Simple Ruleset Language (SRL)

Brownlee describes the Simple Ruleset Language (*SRL*) [4], as a procedural language for creating rulesets for Realtime Traffic Flow Measurement (RTFM). These rulesets, which specify the flows to be measured and how much information should be collected for each, are downloaded to RTFM meters. The RTFM meters use a pattern matching engine to match the downloaded rulesets against attributes extracted from traffic flows to select which flows to monitor. The attributes applied to the traffic flows are specific to network traffic and map to such things as source and destination addresses, port numbers, etc. SRL is not restricted to just traffic metering, but can be useful in any application that involves selecting traffic flows from a stream of packets.

There are two goals of SRL rulesets, which are to identify network packets that are a part of the flow of interest, and then to take some action as a result of the match. The identification of packets is done using IF statements. Actions that are available include the ability to save flow identification attributes, and to keep statistical data about the attributes that are saved.

Figure 9 is an example that counts only TCP/IP packets where the destination port is telnet while saving the source and destination address pair for each packet.

```

#
# Classify IP port numbers
#
define Ipv4 = 1;      # Address Family number from RFC 1700
define telnet = 23;   # Well-known Port numbers from RFC 1700
define tcp = 6;       # Protocol numbers from RCF 1700
#
if SourcePeerType == Ipv4 save;
else ignore;          # Not an Ipv4 packet
#
if (SourceTransType == tcp && DestTransAddress == telnet)
    save, store FlowKind := 'T';
#
save SourcePeerAddress /32;
save DestPeerAddress /32;
count
#

```

Figure 9. SRL Ruleset to Identify and Count Telnet Packets. After [4]

3. Summary of Network Policy Languages

Table 1 summarizes the languages from Sections 3 and 4 that are used to represent network policies or support mechanisms for enforcing policies. Policy-based routing protocols from Section 2 are also represented in the table. The columns of this table represent criteria believed to be useful in comparing the various languages. The “Support Automated Conflict Detection” column refers to the ability to recognize different types of policy conflict as well as the ability to provide a flexible means to resolve conflicts. The column labeled “Suitable for Integrated Services” takes into account the ability to efficiently support Integrated Services. An entry with a “Medium” value signifies that the language can represent a path through the network, but that multiple policies have to be combined to do so. If a column had received a “High” value, then the policy language can represent a path in a direct and intuitive manner, and a policy can be applied directly to that path. The benefits of a “High” value are that policies that

must be associated with all the nodes along a path can be represented with just one statement. This greatly reduces the number of policies statements that a domain must maintain. The "Suitable for Access Control" column refers to the ability to permit or deny access based on policy. The capacity to establish a path through a network and restrict access to that path, is of great importance from a security point of view. The column labeled "Target Architecture" refers to the storage location of the policies. A "Distributed" value means that policies are stored throughout the network, perhaps on individual devices. A centralized value refers to one, or just a few locations where all the policies are located. Having a "Centralized" location is beneficial when trying to detect conflicting policies. The last column, "Ease of Representing Network Policies", takes into account the ability of a user to intuitively represent a policy with the language. Targeting the Path-based Policy Language (PPL) to the group of individuals responsible for representing policies defined with natural language and entering them into a central repository, the author believes the more abstract and closer to natural language, the easier they will be understood. Although these individuals may be versed in formal logic representation, it is believed the majority will be more comfortable with an abstract rule-based language. The more abstract the language and closer to a natural language the higher the value in the column. The greater number of details that have to be specified, the lower the value assigned

Language	Support Automated Conflict Detection	Suitable for Integrated Services	Suitable for Access Control	Target Architecture	Ease of Representing Network Policies
Policy Term	Low	Medium	High	Distributed	High
PFDL	Medium	Medium	High	Centralized	High
RPSL	Low	Low	High	Centralized	Medium
PAX	Low	Low	High	Distributed	Low
SRL	Low	Low	High	Distributed	Low
Policy-based Routing	—	High	High	Distributed	—

Table 1. Summary of Languages that Represent or Support Network Policies

Although the languages represented in table 1 contain many features, none of these languages individually contain all the features provided with PPL. Two major features not adequately addressed by any of these languages are the ability to specify a complete path through a network, and the automatic detection of conflicting policies. The policy-based routing protocols, which are summarized in Section 2, are not concerned with the language used to represent network policies, but instead concentrate on supporting policy-based routing. As a result, columns that have no relevance to these protocols are filled with a "—".

D. LOGIC REPRESENTATION OF POLICIES

This section discusses research using formal logic to represent and detect conflicting policies. The policies represented in this section are more general and involve the use of natural languages, which tend to be ambiguous, to represent policies ranging from management to security.

There has been a great deal of research on the topic of formal representation of policies. Much of this research has been in the area of representing security policies, and the more general problem of translating ambiguous natural language policies into some type of formal representation. Creating an unambiguous language to represent network policies avoids the problems associated with natural languages. PPL focuses on the grammar needed to represent a wide range of network policies while still being manageable enough to provide a process for automatic conversion into a logical representation. Once the policies are in a logical representation, the utilization of methods already developed from research in this area are used to provide a means for checking the consistency of multiple policies.

1. Analyzing Consistency of Security Policies

In Analyzing Consistency of Security Policies[6], the development of a methodology for reasoning about properties of security policies is discussed. The authors view a security policy as a specific case of regulation, where a regulation defines what actions an agent is permitted, obliged or forbidden to perform. With this methodology a system is made up of agents which can perform some actions on some objects. The main focus in this paper is the ability to perform

consistency checks (e.g., check for conflicting situations) on the system, and to have the ability to query a regulation to know which norms apply in a given situation.

To create an unambiguous representation of security policies, the authors use formal logic. According to Chovly and Cuppens the advantage of a representation based on formal logic is the ability to precisely define the axioms⁷ to reason about a regulation. With policies defined by axioms, tools can now be developed to check the system regulation for consistency.

Rather than associating norms (i.e., permissions, obligations, and prohibitions) with individuals, roles are created with these attributes and then individuals are associated with these roles. The individual inherits the norms associated with a role when the individual is playing that role. A conflict can only exist when an individual is playing different roles at the same time, because of an assumption in their research that norms within a role are conflict free.

To resolve conflicts when an individual is playing multiple roles, an ordering is applied when roles are merged. The order represents a priority between them and the order is assumed to be total.

Tools written in Prolog were developed which checked the consistency of the security policies as well as an algorithm for solving conflicts when an individual is playing different roles at the same time. These tools designed to generate consequences of a given set of clauses which belong to a given language and which satisfy a given condition.

2. On the Axiomatization of Security Policies: Some Tentative Observations About Logic Representation

In [20], Michael et. al. add an intermediate step to the traditional approach of translating natural language security policies into their axiom representation. Once the policies are in axiom representation, automated reasoning systems are used for the detection of conflicts. Errors in the translation into axiom form can lead to unidentified conflicts, and incorrect proofs when indeed there is a conflict.

An object-oriented approach is introduced to model the security policies using extended entity-relationship (EER) diagrams. The final axioms of the security policies are then derived from the diagrams rather than directly from the natural language representation. The premise was

⁷ A proposition deemed to be self-evident and assumed without proof

that overall logic rule formulation is simplified in a model-based approach by capturing many of the rules in the structural model.

A case study comparing the two different approaches, model-based and no pre-structuring, produced results that appear to support a premise that fewer structuring errors are made with the model-based approach. A limitation of the model-based approach is that potential queries which might reveal conflicting security policies may be prevented.

3. Policy Hierarchies for Distributed Systems Management

In [22], a policy hierarchy is formed by refining general high-level policies into a number of more specific management policies. This derivation can be performed by refining the goals, partitioning the targets that the policies affect, or delegating the responsibility to another manager. The main motivation for understanding hierarchical relationships between policies is to determine what is required for the satisfaction of policies. If a high-level policy is defined or changed, it should be possible to decide what lower-level policies must be created or changed. The ultimate goal in [22] is to be able to specify high-level policies and automatically generate the lower-level ones.

The goal of policy hierarchy analysis is to determine whether:

- The collected lower-level objectives will completely achieve the higher-level objective which they purport to refine.
- There is conflict between the objectives.
- There is a imperative policy, with a subject, for each objective. An imperative policy gives an agent the imperative to carry out an action. In most cases this implies obligation.
- There is an authority policy which empowers the subject to achieve the objective. An authority policy provides an agent with the legitimate power to perform an action.

4. Conflicts in Policy-based Distributed Systems Management

In [17], policies are used as a means to specify the management behavior of a system, without coding the behavior into the manager agents. Lupu and Sloman focus on techniques and tool support for off-line policy conflict detection and resolution. Two types of policies, authorization and obligation, are addressed in this research. Authorization policy specifies what

activities a manager is permitted or forbidden to perform on a set of target objects. Obligation policies specify what activities a manager must or must not do to a set of target objects and essentially defines the duties of a manager.

Conflicts can arise in a set of policies, but it is not always desirable to eliminate the conflicts by rewriting the policies or changing the membership of the domains to which policies apply. As automated managers cannot enforce conflicting policies, Lupu and Sloman suggest a precedence relationship must be established between the policies in order to resolve the conflicts. Four types of policy's priority are addressed:

- Negative policies always have priority : negative policies take precedence over positive ones.
- The assignment of explicit priorities : policy 1 has priority over policy 2 which has priority over policy 3 ...etc.
- Distance between a policy and the managed objects : priority is given to the policy applying to the closer class in an inheritance hierarchy. For example, a computer science (CS) department is a subclass of a university. If a student is in the CS department, policies of the CS department will override those of the university when a conflict exists.
- Specificity related to domain nesting : a particular case of distance between policies, this principle is that a more specific policy (i.e., a policy applying to a sub-domain) refers to fewer objects so overrides more general policies applying to an ancestor domain.

Lupu and Sloman developed a prototype conflict detection tool that currently detects overlaps between policies and optionally applies domain nesting precedence. The function of the detection tool which is analogous to compile-time type checking for a programming language in that it reduces run-time errors and detects specification errors.

A notation is used to represent policies that is precise and can be analyzed for conflicts using automated tools, but it is not based on a well-known logic. In this system an administrator creates and modifies policies using a policy editor. Checks are made for conflicts, and if necessary policies are modified to remove the conflicts.

5. A Formal Process for Testing the Consistency of Composed Security Policies

In [21], Michael presents a formal process for testing the logical consistency of composed security policies. The introduction of a structural model is made to represent relationships between security policies, and axiomatizes the policies so that relationships constructed in the model are preserved and made explicit in a logic model. This logic model is then used for deductive proofs of policy consistency. Michael states that problems arise in correctly defining, evaluating, and mapping policies onto procedures and that a structural model reduces these types of gaps.

OTTER, an automated first-order resolution-style theorem prover is used to detect logical contradictions between the axioms in the logic model.

E. CHAPTER SUMMARY

In this chapter, several policy languages were reviewed, as well as techniques used in policy conflict detection and resolution.

The policy languages fell into two major categories: the abstract network policy languages and bit-level traffic flow languages. Each of these languages were designed to address particular areas of network policy. Among these areas are:

- Differentiated Services
- Integrated Services
- Quality of Service
- Traffic Identification

The techniques reviewed for policy conflict detection are not focused on network policy, and address the difficult problem of translating ambiguous natural language policies into a formal representation. It is clear that there is a need for network policy conflict detection, evident in the fact that many of the network policy languages surveyed have features to resolve conflicts once they are detected. What is lacking is an automatic method for checking individual network policies that are composed together to satisfy an overall goal.

The Path-based Policy Language (PPL) introduced in this thesis encompasses as many of the features addressed in the previous languages as possible, as well as providing a means for testing policies for consistency. Table 2 states the major goals in developing a network policy language,

which is more suited toward what network policy implementers are accustomed to: a rule-based representation more closely associated with a computer programming language. Taking a path-based approach will enable us to establish policies that will be based on path, like Integrated Services, as well as non-path based policies which are more suited toward Differentiated Services. The use of a wild card character enables us to describe policies based on the concepts of Differentiated Services or best-effort traffic.

III. PATH-BASED POLICY LANGUAGE (PPL)

A. INTRODUCTION

This chapter will introduce the Path-based Policy Language (PPL). This new language, is intended to solve and/or alleviate many of the deficiencies of the languages which were discussed in the previous chapter. PPL has the ability to represent network policies unambiguously, providing support to heterogeneous networks for which the networks are controlled using explicit policies. Policies required by both path and non-path based traffic flows are supported with PPL as well the ability to support conflict detection and resolution with the use of formal logic.

B. GOALS

PPL is designed to support policies that can be applied to Differentiated Service, Integrated Service as well as Multiprotocol Label Switching models proposed by the IETF. Table 2 states the goals that PPL had strove to attain.

PPL Goals	
1	create a path-based representation of policies flexible enough to support both path and non-path based traffic flows.
2	represent network policies in an unambiguous way
3	be abstract enough to cross device and manufacturer boundaries
4	detect and resolve conflicts between policies
5	support the dynamic aspect of networks

Table 2. Summary of the goals of PPL

1. Path & Non-Path Based Traffic Flows

To support models such as Integrated Services (path based) as well as Differentiated Services (which may not be path-based) PPL must be flexible. Providing an absolute path consisting of the links the traffic must take will provide greater control over traffic flows and provide easier support to integrated services. A less specific policy may only need to provide

source and destination nodes in its configuration, or perhaps just the specification that all traffic of lets say file transfers, must be forwarded through a specific node acting as a firewall in an edge router.

At the same time PPL will have to support non-path based policies such those associated with best effort service and possibly differentiated services. These types of flows classify and forward packets through the network based on the classification the packet receives. This enables preferential treatment to be given to particular service classes of data on a per node basis.

2. Abstract

Our language is abstract enough to support multiple vendors and devices. Providing a language that is too specific will demand constant updates to the language, as well as to software on the vendor's devices. The way these policies are enforced with a vendor's equipment is not as important as the fact that they can be enforced. Each specific device should be able to implement the policy rule without specifying the exact details.

3. Unambiguous Policies

Our path-based language represents network policies in an unambiguous way. This feature allows us to detect policies that are in conflict as well as create a stable network environment. A policy that can be interpreted multiple ways will prevent a clear understanding of the network configuration and can potentially hide conflicting policies from the network manager.

4. Detect & Resolve Conflicts

Several languages summarized in the previous chapter provide features that support the ability to resolve policy conflicts, but fall short in providing a mechanism to detect such conflicts. PPL provides the ability to detect and resolve conflicts between policies using the translation of policy rules into formal logic. The compiler which works hand and hand with the Prolog programming language interpreter, evaluates the network policies and returns information about which policies are in conflict.

5. Dynamic Policies

The state of a network is rarely constant and therefore a policy language should be able to represent network policies based on dynamic factors as well as constant ones. PPL not only supports static policies such as attributes based on traffic class or network address, but also policies based on dynamic factors such as packet delay and packet loss rate.

To support a dynamic network, PPL provides the ability to react to dynamic conditions of the network. For example, rather than just denying or permitting network traffic on a link that is suffering network delay, perhaps it would be better to compromise and lower the priority of a traffic class while the delay is occurring. Once the problem in the network delay is resolved, a second policy might support the upgrading of the priority of that same traffic class. This is all possible by allowing the policy maker to specify which messages, detailing the current state of the network, to collect and react to.

C. POLICIES

1. Support of Integrated Services

Integrated services provide QoS assurances on a per-flow basis. This per-flow concept allows for the allocation and reservation of enough resources along a path to support a QoS request for a flow of packets. Integrated services using a signaling protocol such as RSVP can request resources along a path in a dynamic nature. If a path can not support the QoS demands of a request, then the particular path will be rejected.

This ability to reserve and control the amount of resources allocated along a path through the network, or more importantly not to over allocate resources on the path, provides a great advantage when trying to support QoS.

The importance of this per-flow concept is why PPL was based on the concept of path. The third element in a policy rule represents the path(s) effected by the policy. This path attribute has many flexible and powerful features.

- **Multiple paths:** Assigning the same policy to multiple paths in the network is easily accomplished with PPL. This is carried out with a comma separated list as the third argument. The policy below demonstrates this feature by assigning the same policy which denies access to a block of IP network addresses to multiple paths.

Policy1 net_manager {path1, path2} {*} {hostIP = 161.1.*.*} {DENY}

- **Wildcard character support:** The path argument in a PPL policy also supports the wildcard character. This is extremely useful when you have a policy that should be applied to all the paths in your network. One example would be to assign a maximum *hopcount* value to packets in your network. The policy below shows such a policy.

Policy2 net_manager {*} {*} {hopcount > 19} {DENY}

- **Path:** PPL supports the defining of paths and assigning attributes to these paths such as bandwidth capacity and dynamic messages supported. Once a path is defined, it's name can be used in the path argument of a PPL policy as shown in the following example.

```
define path example_path {Node_1, Node_2, Node_3};  
Policy3 Net_manager {example_path} {*} {*} {permit};
```

- **Link:** A link is defined in PPL by identifying the two nodes of the network that create the link. Since a link is a simple path of just two nodes, it can also be used in the path argument of a PPL policy, as the following example demonstrates.

```
define link example_link <Node_1, Node_2>;  
Policy3 Net_manager {example_link} {*} {*} {permit};
```

- **Node:** Typically a single node would not be considered a path or link, but to support scalability, PPL provides the ability to assign a policy to a node. This concept is better demonstrated using Figure 1. At level 2 the network is not concerned with the details of the Naval Postgraduate School or the DARPA networks from level 1. All that is of concern are the policies that are supported in those lower level networks. This allow policies to be assigned to nodes representing networks to support QoS efforts at a higher more abstract level.

For clarity, there is a difference between a path defined as {<*,NPS,*>} and the path {NPS}. The first represents all possible paths that include the node NPS at some point. The second represents just the node NPS, which again might represent an entire sub-net at a different level in the network hierarchy. The following is an example of a policy rule that is expressed over a node.

```
define path example_node Node_1;  
Policy3 Net_manager {example_node} {*} {*} {permit};
```

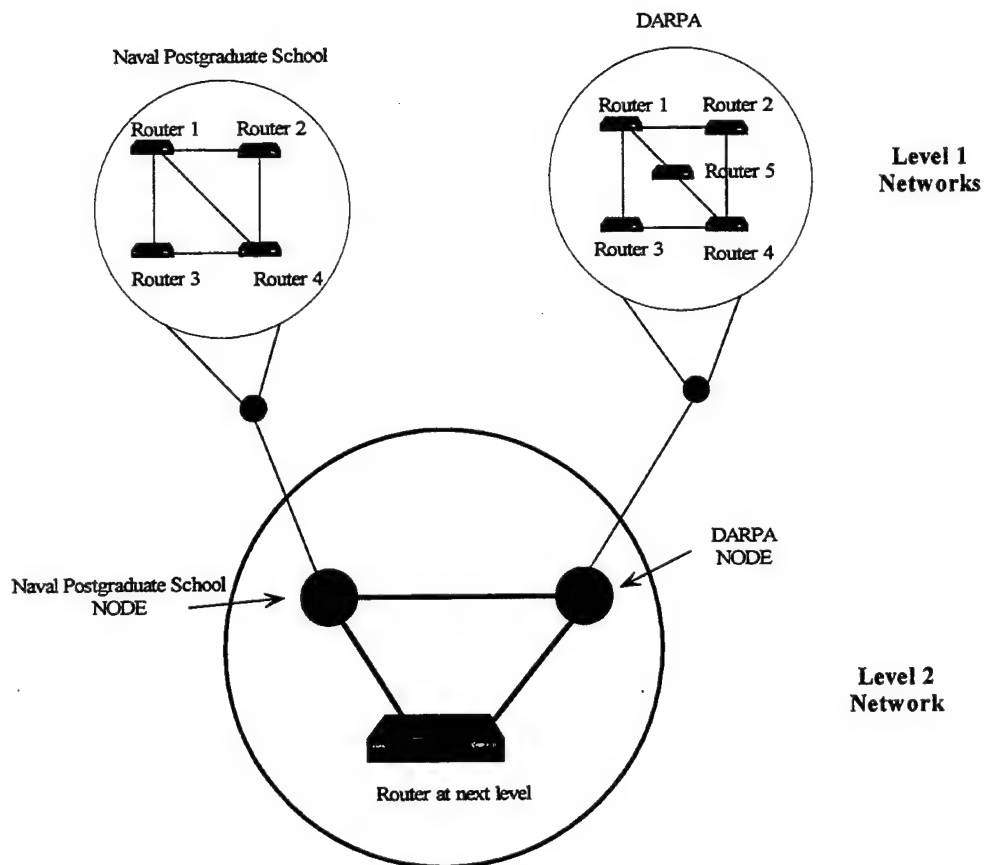


Figure 10. Node Representing Sub-Network

Integrated services are not only based on a path through the network, but certain conditions must also be provided along the path to be acceptable. To accommodate this feature PPL provides a *conditions* argument in a policy statement which maps to the fifth element of the policy rule. All the conditions listed in a policy rule must be met before the actions associated with the policy are executed. The following examples show how the *conditions* argument can be used.

```
Policy6 net_manager {path1} {traffic_class == {video}} {time >= 0800, time <= 1600}
{DENY};
```

```
Policy7 net_manager {path1} {traffic_class == {video}}
{jitter() < 3 msec, delay() < 2 msec} {PERMIT};
```

Policy6 represents a policy that denies the use of a path to video traffic between the hours of 8:00am and 4:00pm, normal working hours.

Policy7 represents a policy that permits video traffic over a path as long as the jitter() and delay() on the path are acceptable. In this example jitter() and delay() are *messages* that provide dynamic feedback on the current conditions of the path.

Action_items is the last element of a PPL policy rule. Typically this parameter will either DENY or PERMIT packets of a traffic flow depending on the type of traffic contained in that flow, or as a reaction to dynamic feedback from network devices. For example, policy 8 below will DENY packets originating from the IP network 141.1.*.* on the link between NPS and DARPA.

```
Policy8 net_manager {NPS_DARPA}{*} {hostIP == 141.1.*.*} {DENY};
```

Policy 9 below will also DENY traffic, but this time as a result of network delay. Although the policy is reacting to the dynamic nature of the network and may even eliminate the current network delay, it is reacting in a very rigid manner. Rather than denying all video traffic, perhaps a better solution would be to lower the priority of the other traffic types.

```
Policy9 net_manager {NSP_DARPA} {traffic_class == {video}} {delay() > 20 ms} {DENY};
```

This *action_items* element can be also used to dynamically compromise on a network flow's requirements if so desired. For example, instead of totally denying a traffic flow containing video data, perhaps a better solution would be to lower the priority of the video traffic. In this way non-video traffic could be provided better service while the video traffic is still allowed to flow, but at a degraded rate. This feature provides the ability for a network flow to throttle back for the general good of the network. The following policy represents such a requirement.

```
Policy10 net_manager {DARA_SPAWAR} {traffic_class == {video}}  
{loss_rate() > 20%} {priority := 20};
```

2. Support of Differentiated Services

In differentiated services, packets are marked differently to create several packet classes. Packets in different classes receive different services[37]. This concept allows for preferential treatment of organizations who are willing to pay more for a particular service class. This classification of traffic may be a single service class, for an individual company for example, or may contain several classes which differing requirements. A telephony application will require a class of traffic that can provide low-delay and low-jitter whereas an email service does not require such timely delivery.

Differentiated services is essentially a relative-priority scheme where individual packets are classified and marked for forwarding. The handling of packets based on its classification marking can provide several classes of traffic in a network.

To support policies based on differentiated services, a PPL policy rule contains a *target* field represented by the fourth element in the policy statement. To utilize this field, a traffic class has to be created prior to its use in a policy. Some examples of defined classes of traffic are given below.

```
define class traffic_class {data, video, voice};  
define class traffic_priority {high, med, low};  
define class user {faculty, student, staff};
```

In these defined classes, *traffic_class*, *traffic_priority*, and *user*, predefined values are assigned to each class. For example if a path supported *traffic_class*, then acceptable classes on that *path* would be *data*, *voice*, and *video*. A policy may be defined to support multiple classes of traffic on a *path*. When such a situation is desired a list of classes is presented in the policy statement. When more than one class is given in a statement the effected *path* will support any of the provided classes. For example, consider the following policy.

```
Policy1 net_manager {*} {traffic_class = {data}, {traffic_priority = {high, med, low}}  
{*} {PERMIT};
```

The fourth element representing the classes of traffic permitted, indicate that *data* traffic is permitted, as well as either *high*, *med*, or *low* priority traffic. When multiple classes of traffic

are presented in a policy statement as a comma separated list, the classes are logically OR'd together.

3. Support of MPLS

Multi-Protocol Label Switching (MPLS)[36,37] is a forwarding scheme that provides fast packet classification as well as the ability to support efficient tunneling. At the edge router of a MPLS-capable domain the network protocol packets are classified and routed based on the information of their network headers as well as routing information stored at the label-switching routers. To make the forwarding efficient, an MPLS header is inserted between the link layer and network layer of every packet traversing the MPLS domain. The MPLS header contains a 20-bit label, a 3-bit Class of Service field, a 1-bit label stack indicator, and an 8-bit Time to Live field. The MPLS header allows label-switched routers within a domain, to examine only the label in forwarding the packet. When the packet leaves the MPLS domain, the inserted header is removed. These features make MPLS very useful for traffic engineering.

One of the most useful features of MPLS is the ability to specify an explicit route through the MPLS domain, in essence set up a path. Since PPL establishes a path through a network based on a policy describing the network, it naturally supports MPLS as well. The *target* and *conditions* fields of a PPL policy rule can be used to do the classification of the packet, and the path associated with the policy can be used to establish routing information in the label-switched routers.

4. Support of Static Policies

Policies based on static information can be easily represented with PPL. These policies are based on information contained in the network packets themselves and do not rely on the dynamic aspects of the network such as delay, jitter, and loss rate. Static policies are essential and can be used to establish classes of traffic as seen in differentiated services, or in establishing a path through the network as is seen with integrated services and MPLS. Examples of policies based on static information are given below.

```
Policy1 net_manager {link1} {hostIP == 206.3.4.*} {DENY};
```

This policy will deny packets from a particular network address (206.3.4.*) to pass through link1.

```
define path video_path {<node1,node2,node3>;
```

```
Policy2 stone {video_path} {traffic_type == {video, voice}} {*} {PERMIT};
```

Policy2 establishes a path through a network that permits both video and voice traffic. If no other policies are associated with the nodes that make up this link, then video and voice are the only types of traffic that are permitted. This result is because PPL implements a default action of 'deny' unless explicitly 'permitted'.

D. MESSAGES

Dynamic policies require feedback from network devices to keep apprised of the current state of the network. To support dynamic policies in PPL, the concept of *messages* was introduced. A *message* is used to transcend the issues associated with platform-dependent network technology, allowing us to specify functionality rather than deal with implementation details. A *message* represents an information update about some measurable attribute of a network. For example a *message* identified as *delay()* might provide the time required to travel from point A to point B in a network in milliseconds. To maintain abstract, PPL, is not concerned with the definition of the *message*, just that the *message* will be supported on the path, and used consistently. Below is an example of a policy that supports dynamic feedback with the use of a *message*. Before the policy can use the *message*, it has to be associated previously with the *path*. The *define path_param* statement associates the *delay()* *message* with the *path* *NPS_INTERNET*, as well as defining the *bandwidth* associated with the link. The policy statement itself uses the *delay()* *message* as an argument in the *conditions* element of the policy to deny access to *video traffic* when the *delay()* is too great.

```
define path_param NPS_INTERNET {BW := 500 MBPS, delay()};
```

```
Policy3 net_manager {NPS_INTERNET}{traffic_class == {voice}}  
{delay() > 20 ms} {DENY};
```

During the policy testing phase of the PPL compiler, a policy utilizing the feedback of a *message* will also be tested for conflicts. This test involves the verification that all links required to compose the *path(s)* associated with a policy, also support the *message*.

E. PPL POLICY FORMAT

The format of PPL is summarized in Figure 11. A policy rule consists of six elements. The formal definition of the grammar is defined in appendix A.

PolicyID userID {paths} {target} {conditions} {action_items}

policyID- unique policy identification token

userID - user ID of policy creator

paths - network paths the policy affects

target - target class of network traffic (items are OR'ed)

conditions - any global conditions (items are AND'ed)

action_items - for setting parameters (e.g., policy priority), explicit deny or permit, etc.

Figure 11. Summary of PPL format

We illustrate the capabilities of PPL through the use of several examples below. Example 1 shows the ability to specify an explicit path for a traffic flow. Examples 2, 3, 4, 6, and 7 use the wild card character '*' to specify partial *paths* for traffic flows. In example 6, the use of '*' places no restrictions on the *path* the traffic may take.

In example 4 a policy is represented that will make a compromise when certain network conditions are met. This compromise feature provides the ability to throttle back network flows for the general good of the network.

Example 1: Policy 1 net_manager {<1,2,5>} {class = {faculty}} {*} {priority := 1};

This is a rule which states that the path starting at node 1, traversing to node 2, and ending at node 5 will provide high priority for faculty users.

Example 2: Policy2 stone {<*,2,*>, <*,4,*>} {*} time >= 1600, time <= 0800} {deny};

This rule states that all traffic will be allowed to traverse through nodes 2 and 4 during non-working hours. Unless granted by another policy, traffic will not be able to traverse through nodes 2 and 4 during working hours.

Example 3: Policy3 `net_manager {<*,5,*> {*} {hostIP = 131.1.*.*} {permit};`

All hosts with a network address starting with 131.1 will be permitted to traverse node 5. Having the ability to restrict groups of network addresses as well as individual network addresses is also a part of PPL.

Example 4: Policy4 `xie{<1,*,*,5>}`

`{traffic_class = {video, voice}} {used_bw() <= allotted_bw(),
allotted_bw() = 50MBPS, loss_rate() > 40%} {allotted_bw := 40MBPS};`

This policy shows the ability for compromise. Voice and Video traffic are provided with an allotted bandwidth of 50 Mb/s, but when the network loss rate is greater than 40%, a compromise will be made to lower the allotted bandwidth to 40 Mb/s.

Example 5: Policy5 `net_manager {*} {traffic_class = {data}} {*} {priority :=10};`

All data traffic will be assigned a priority level of 10. Assume that there are three classes of traffic for this example, voice, video, and data. This allows for providing higher or lower priority to certain classes of traffic. In this case the priority might affect the order of packets being dropped from queues at network devices during times of congestion.

Example 6: Policy6 `Betty {<1,*,5>} {traffic_class = {accounting},
day != Friday} {*} {priority := 5};`

On all paths from node 1 to node 5, accounting class traffic will be lowered to priority 5 unless it is a Friday. In this policy the `action_items` field is used with temporal information to influence the priority of a class of traffic. It might make sense to have this feature when departments of a company need more network resources to accomplish their jobs.

Example 7: Policy7 `net_manager {<1,*,5>} {traffic_class = {student},
userID = Gary} {*} {deny};`

On all paths from node 1 to node 5, deny access to network traffic from user Gary who is in the student traffic class. This policy shows that PPL can provide control at a very small granularity level. In this case the policy affects only a single user in a particular

class of network traffic. It could have easily been modified to provide certain times the days when it was in effect as well.

F. CHAPTER SUMMARY

In this chapter, the format and features of the Path-based Policy Language (PPL) were introduced. The six element policy rule supports the combined features of previous network policy languages as well as the following new features:

- Path-based representation
- Support of both static and dynamic policies
- Support of future traffic classes
- “Message” support for existing and new network measurements
- Ability to scale well in large networks
- Policy conflict detection

Several policy examples were presented to show the power and flexibility of PPL, and to demonstrate the usefulness of such a language by expressing realistic network policies.

IV. POLICY CONFLICT DETECTION AND RESOLUTION

A. INTRODUCTION

One major contribution PPL makes is the ability to detect and resolve policy conflicts. Having conflicting policies without the means to detect and resolve them is probably worse than having no policies at all. An obvious example involves security access policies. Having policies defined to restrict access to network resources can build a false sense of security, when a rogue policy conflicts with existing policies to provide access to those same restricted resources. Believing that the restrictive policies are working may prevent network administrators or security personal from verifying the protection, which can have very serious consequences.

As was reported in chapter II, formal logic has been used in previous efforts such as with security policies and general management policies to detect conflicts between composed policies. One of the major problems with these previous efforts has been the ability to translate the natural language policies into their formal logic equivalent representation. Efforts were made during the creation of PPL to represent network policies without the inherent ambiguity of a natural language.

Once the language (PPL) was created, it was then possible to develop a compiler that would parse the network policies and verify their correctness according to the grammar specified in appendix A. If all the policies were both syntactically and semantically correct, then a formal logic representation of those policies was created.

Figure 12 shows the overall process flow starting at the acceptance of a PPL configuration file, which contains network construction information and network policies, to the final output file which contains a list of policies which created conflicts. Once the PPL compiler verifies the syntax and semantics of the input configuration file, an output file consisting of logical facts is generated from the configuration file. This logic file is the starting point of a three stage process that manipulates and refines the data that eventually results in the final conflict file. Although the three stages could have been accomplished in just one stage, multiple stages provided an incremental means to verify correctness during development.

To create the compiler for PPL, tools designed to aid in the construction of compilers were used. *Flex*, a tool for creating a scanner for a language, was used to scan and return the

tokens required during the parsing phase of the compiler. *Bison*, a tool for writing a parser for a language, was used to create the parser that verified the correctness of PPL network policy rules. Correctness in this sense, refers to policy rules that are both syntactically and semantically correct as specified in the PPL grammar.

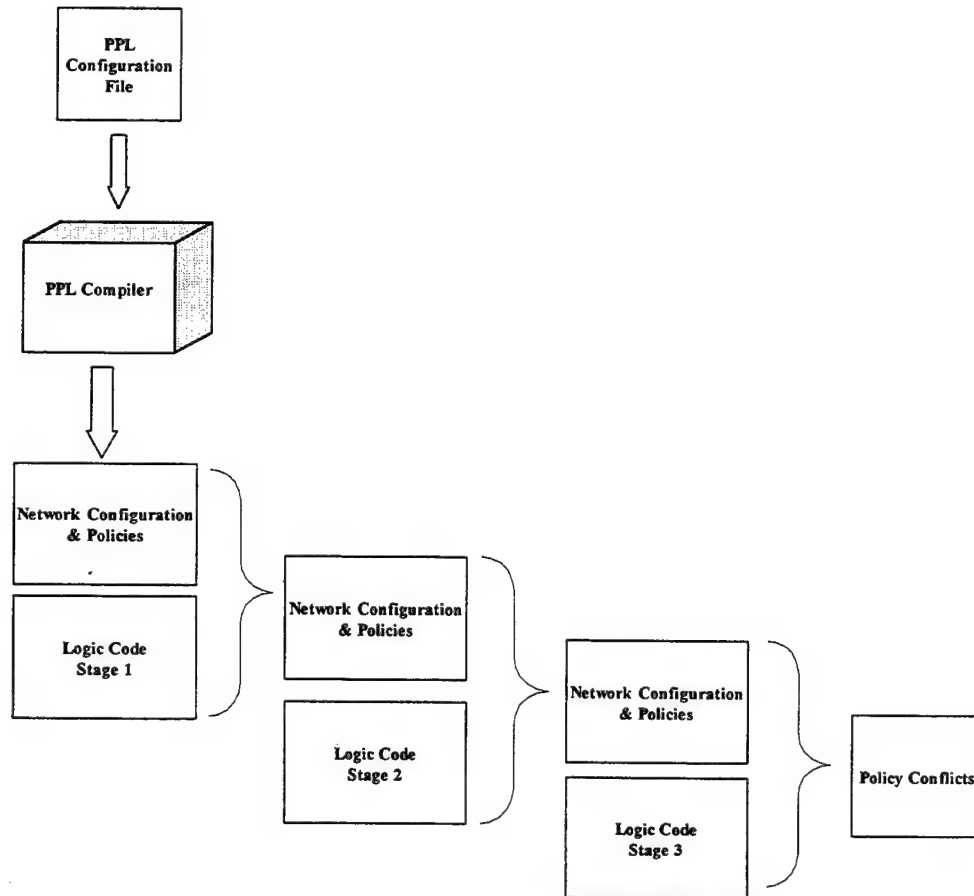


Figure 12. Overall Process Flow

Once the correctness of the PPL policy statements were verified, an output file with Prolog statements that represented the physical description of the network, such as network links, capacity, etc. was created. In Prolog these type of statements are called facts and are used to prove the validity of questions presented to the language interpreter. The actual PPL policy rules were then converted and represented in Prolog and appended to the output file as facts and rules. A rule in Prolog declares things that are true depending on a given condition. Prolog

(*programming in logic*), is a programming language based on first-order predicate logic. The advantage of using Prolog is that what constitutes a conflict can be represented, and then questions can be proposed to the interpreter as to whether a conflict existed. Using mechanisms such as pattern matching, tree-based data structuring, and automatic backtracking, the Prolog program can return a truth value representing an answer to a question, in the case of PPL's conflict tester, "Are there conflicts between any network policies?".

When policy conflicts are detected, a PPL build-in support mechanism will be used in an attempt to resolved the conflict. The support mechanism utilizes the creators of each policy to check for a possible precedence relationship. Both resolved and unresolved conflicts are presented to the operator for their review.

The development and test platform used to build the supporting compiler and policy conflict tester, was a Media On Pentium II computer running the Microsoft Windows 98 operating system. Although the underlying operating system was Windows 98, the actually development environment was the BASH programming shell provided by Cygwin. Cygwin tools are ports of the popular GNU development tools and utilities for Windows 95, 98, and NT. They function by using the Cygwin library which provides a UNIX-like API on top of the Win32 API. The Prolog compiler used was SWI-Prolog, developed at the University of Amsterdam, and is targeted primarily at research and education. The details of the hardware and software used are listed below.

Hardware Description

- CPU: 300 MHz Pentium II-MMX
- RAM: 64 MB SDRAM

Software Description

- Operating System: Microsoft Windows 98
- Shell: GNU BASH version 2.01.1(2)-release (i386-pc-cygwin32)
- C Compiler: Cygnus gcc version 2.7-B19
- Flex: GNU flex version 1.25
- Bison: Cygnus bison version 2.5-B19
- File Utilities: GNU fileutils 3.16
- Prolog: SWI-Prolog (Version 2.8.2)

The source code for the PPL parser, written in Bison with imbedded "C", is listed in appendix D. Appendix E contains the Prolog source code for the three stages of conflict detection and resolution.

B. DETECTING CONFLICTS

There are multiple elements to a PPL policy rule and consequently there are multiple types of conflicts that can exist. The key to all conflicts is the *path* or *paths* associated with a policy rule. PPL is based on *path*, and this *path* is used as the first step in identifying a conflict. Without the overlap of at least one *path segment* between policies, there will be no conflict.

In PPL a conflict is defined as follows:

Given a set of policies P , a conflict exists if for any 2 policies $r, s \in P$, all of the following hold:

1. Physical paths of r and s overlap on at least one path segment;
2. Time and network conditions specified in the *conditions* elements of r and s overlap;
3. There is a *target a* which is permitted in r , but denied in s (or conversely).

Based on the definition above, the following general steps are implemented and used in the conflict detection process of the PPL policy tester.

1. Find all policies that contain overlapping *paths* (physical).
2. Prune the results from step 1 to only contain those *paths* that also have overlapping *conditions* (timing, network conditions, etc.)
3. Use the *target & action_item* elements to determine if a conflict exists

Each *path* associated with a policy rule specifies the nodes and links the traffic will follow, but the *conditions* of the policy rule can restrict that flow of traffic to a particular day of the week, hour of the day, etc. This combination of *path* and *conditions* can allow voice traffic on the same *path* that a second policy has denied without conflict, as long as the day and time do not overlap.

When there are indeed overlapping *paths* in a network, the *target* and *action_items* will determine if there is truly a conflict. The *target* specifies the class of traffic that is allowed on the specified *path*, and the *action item*, among other things will either permit or deny that traffic. This ability to either permit or deny traffic is what leads to the fundamental conflict between policy rules.

The remaining sections of this chapter expand on the steps used to detect conflicts with PPL defined network elements and policies.

1. Basic Conflicts

Even before policy rules are processed for conflicting *targets*, *conditions*, and *actions*, a conflict can exist. When *nodes*, *links*, and *paths* are being defined for use in the policy rules, attributes about those elements are specified. These attributes consist of *bandwidth* capacity and the *messages* that will be supported on the *path*. *Bandwidth* conflicts are handled by the PPL compiler in the following fashion. It is obvious in Figure 13, that when the links *Link_1* and *Link_2* are defined with a bandwidth of 100 Mbps each, that there is no way *Path_1* can produce can be defined with a 500 Mbps connection, therefore a conflict exists. This is the most basic conflict based on path and is easily identified.

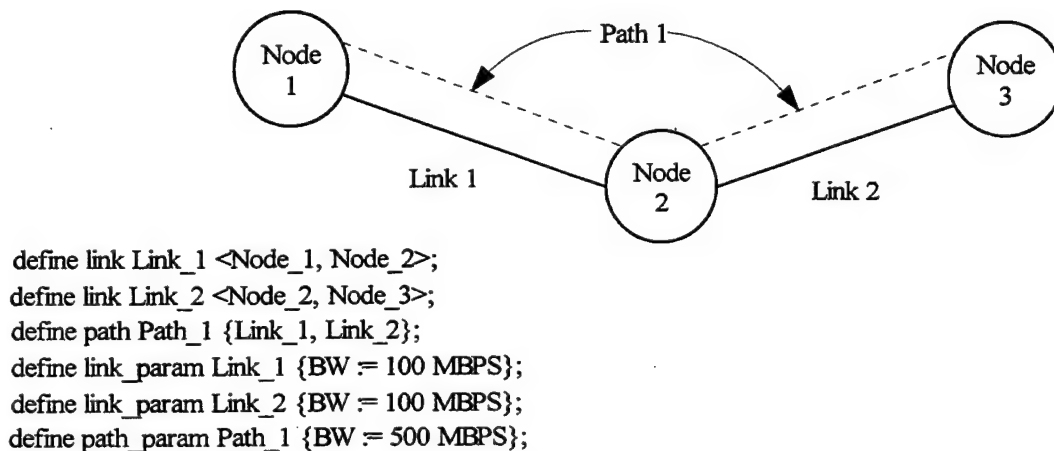


Figure 13. Bandwidth Conflict

Message conflicts are based on the same premise as *bandwidth* conflicts. When a policy rule utilizes a *message* in the *conditional* element, the *paths* associated with this policy must provide the required *message* feedback. The *message* conflict detection phase, verifies that every link needed to construct the effected *paths*, provides the proper *message* support.

2. Expanding Paths

Paths can be specified explicitly listing each node and link along the path from source to destination. When this happens, the *path* is associated with the *PolicyID* and stored as a fact in prolog. Facts about the network representation and policy rules are used to answer questions about policy conflicts in later stages of the compiler.

The use of the wildcard character '*' has many benefits in the representation of a *path* as discussed in earlier. These compressed representations of *paths* must be expanded to check for overlapping *paths*. Figure 14 represents a limited view of a network. When a user wants to represent all *paths* from node 1 to node 4, rather than listing two separate *paths*, $\langle 1, 2, 4 \rangle$ and $\langle 1, 3, 4 \rangle$, the user may instead represent both *paths* with the single *path* $\langle 1, *, 4 \rangle$. This compacted representation helps save space in policy databases. The savings are even greater when there are more than two *paths* from node 1 to node 4. During the process of conflict detection, the *path* $\langle 1, *, 4 \rangle$ will indeed be expanded temporarily to search for overlapping path segments.

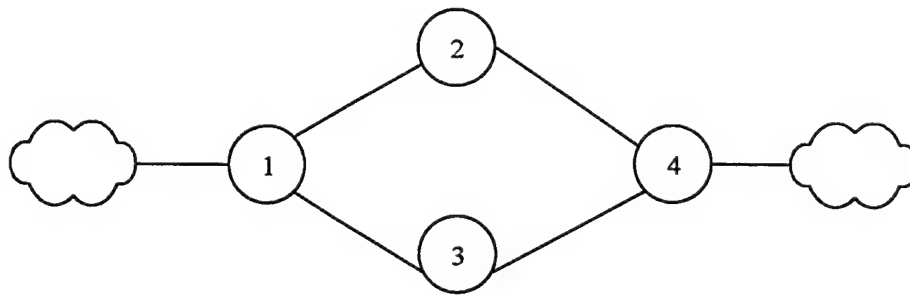


Figure 14. Network View

The *path* element of the policy rule supports the listing of several *paths* independently, as well multiple *paths* with wildcard characters. Examples include $\{\langle 1, 2, 4 \rangle, \langle *, 3 \rangle\}$ and $\{\langle *, 2, * \rangle, \langle 1, 3, 4 \rangle\}$. In all these *path* representations, each *path* has to be expanded and associated with the *policyID* of the rule. These expanded *paths* are used in conjunction with other policy rule elements such as, the *conditional element*, *target class*, and *action items*, to identify conflicts between PPL policies.

3. Conditional Overlaps

The conditional element of a policy rule provides a way to further isolate a *path* associated with that rule. It is possible that two policies apply to the same *path segment* in a network and that the *actions* of those policies will cause a conflicting situation. The conditional element of the policy rule can further specify attributes such as time of day, day of the week, a particular user, etc. The *conditional element* may allow two seemingly conflicting policies based solely on their *paths*, to co-exist without conflict. When several attributes are listed in the *conditional* field, the result will be the concatenation of each. For example, when both time of day and particular user are specified in the *conditional* element, both have to be true to validate the policy. In other words all the attributes in the *conditional* element are AND'd together. The process of expanding *conditional* elements is discussed below. The discussion also includes how the comparison of *conditional* elements from different policy rules may lead to overlapping *conditions*. An overlapping *condition* exists if the attributes of a *conditional* element provide some intersection in their values. For example, if a *conditional* element for a policy specifies that it will only be enforced on week days, and a second policy specifies it will only apply be enforced on Tuesdays, then an overlap occurs on Tuesday. This information about overlapping *conditions* along with the expanded path information is used in future steps of the conflict detection process, which include the *target* and *action items* of the policy rule.

Time is represented with a 24 hour clock in PPL and is used in the *conditional* section of a policy rule to constrain when the rule will be applied. Like all *conditional* elements used in PPL, an overlap of *time* between two policies indicates that there is a potential for a conflict to exist. Only two *conditional* operators are used when representing a *time* constraint, \geq and \leq . These two symbols are sufficient to represent the most common planned usage of *time*, which is specifying ranges of *time* the policy covers, such as between 0400 hours and 0800 hours. The use of \leq and \geq can also be used to simulate other comparison operators such as $=$, \neq , $<$, and $>$. For example, if a policy was only to be enforced at one instance in time such as 0400, this can be specified with the two conditions, $\geq 0359:59$ & $\leq 0400:01$. PPL does not limit the number of occurrences of *time* in the conditional section of the policy rule and therefore several *time* ranges can be applied to the same policy. Table 3 represents the logical outcome used by the PPL compiler to determine if a *time* overlap exists. When more than one time is specified in a policy statement, each of those specific *time* entries will be compared against all the *time* entries in the

second policy. For example, one scenario represented in the table below is: *Policy A* applies between the hours of 0400 and 2359, and *Policy B* applies to the hours between 0300 and 0700. The result of combining these two constraints is that there is indeed an overlap in *time*, specifically between 0400 and 0700. There is no need to have an exact overlap in *time*, as long as there is a partial overlap in *time* the potential exists for a conflict. Also note that *Policy A* does not explicitly identify 2359 as the ending *time* of the condition, but whenever there is no upper or lower limited identified, a default value of 0000 or 2359 is implied. Specifying only the time \geq 0400, therefore implies an upper limit of the 24th hour of the day.

Time Range Policy A	Time Range Policy B		Result of Combined Time Ranges
\geq 0400	\leq 0700	-	Overlap
\geq 0400	\geq 0700	-	Overlap
\leq 0400	\leq 0700	-	Overlap
\leq 0400	\geq 0700	-	No Overlap
\geq 0700	\leq 0400	-	No Overlap
\geq 0700	\geq 0400	-	Overlap
\leq 0700	\leq 0400	-	Overlap
\leq 0700	\geq 0400	-	Overlap
\geq 0400	\geq 0300	\leq 0700	Overlap
\leq 0400	\geq 0300	\leq 0700	Overlap
\geq 0400	\leq 0300	\geq 0700	No Overlap
\leq 0400	\geq 0300	\leq 0700	Overlap
\geq X	\leq X	-	Overlap
\geq X	\geq X	-	Overlap
\leq X	\leq X	-	Overlap
\leq X	\geq X	-	Overlap
-	*	*	Overlap
-	-	-	Overlap

Table 3. Conditional Time Overlap

Also displayed in the table are entries that indicate that when the *time* values are exactly the same, represented with an X, there is by default overlap because the equivalence that occurs with the \geq and \leq operations. As with all the conditional elements, when no time is explicitly declared, it implies that the policy will be active 24 hours a day and therefore will overlap with any other policy rule. An '-' entry in the table below indicates that no *time* was explicitly indicated, and the use of the '*' symbol represents any *time* may be applied.

The built-in *user* attribute can also be used in the conditional section of a policy rule to place constraints on a rule. The *user* attribute allows another way to segregate network traffic. The *user* represents an identifier that uniquely identifies a particular user on the network. For example, the policy below indicates that data traffic will be denied if the *user* associated with the traffic is equal to *gnstone*.

Policy_user net_manager {*} {traffic_class = {data}} {user = gnstone} {deny}

When two policies are being compared where at least one of those policy rules contains the user attribute, there is a chance of an overlap of coverage between those policies and continued conditional comparisons are warranted to verify consistency. Table 4 represents the results of the evaluation of *user* attributes. The only operators allowed with the *user* attribute are the comparison operators $=$ and \neq . Since no ranges or ordering are allowed with *users*, the use of the operators $<$, $>$, and all variations of those operators are meaningless. When no *user* is specified in the *conditional* element of a policy rule, the default action is to allow all users. When only one of those policies contains the *user* attribute, then again there exists an overlap. This is a result of the fact that no *user* restriction on a rule implies all *users* are specified. In Table 4 two comparison results are also marked with an asterisk. The asterisk is used to identify that fact that more than two *users* are assumed to exist on the network. For example, if one policy specified *user* \neq *gnstone* and a second policy specified *user* \neq *lundy*, then the comparison will identify an overlap. Assuming there are other *users* besides *gnstone* and *lundy*, this is the proper action. If this is not the case and only those two *users* exist in the network, then the policies are meaningless because neither will ever be enforced because the condition will never be satisfied.

User A	User B	Result of Combined User Resolution
= gnstone	!= lundy	Overlap
= gnstone	!= gnstone	Conflict
= gnstone	= lundy	No Overlap
= gnstone	= gnstone	Overlap
!= gnstone	!= lundy	Overlap*
!= gnstone	!= gnstone	Overlap
-	!= gnstone	Overlap*
-	= gnstone	Overlap
-	-	Overlap

Table 4. Conditional User ID Overlap

The *address* attribute used in PPL is based on the format of the Internet Protocol (IP) used today in the Internet. Other *address* formats could also be added in the future, including IP version 6. During the comparison of the *address* attributes, two separate evaluations are performed. The first is whether an overlap exists between the *addresses* represented, and the second is whether the use of the operators ' \leq ' and ' \geq ' applied with the first decision result in an overlap.

An IP address is usually written, and is represented in PPL with dotted decimal notation. In this format, four decimal values ranging from 0 to 255 are concatenated together using a '.' as a separator. An IP address contains both a *network part* and a *host part* with some number of bits used to identify the network, and the remaining number of bits used to identify the host. This subdivision of bits will vary depending on the class of the address, but there is no real need to totally understand how the addressing works to proceed. The idea is that different granularities can be used to isolate network traffic. It was documented earlier how an explicit individual can be isolated using the *user* attribute. The *address* attribute allows for the enforcement of policies at the individual machine level, or specifying a group of devices for enforcement. A typical IP address will look like 131.120.10.104. To implement a range of IP addresses, PPL implemented a

simple pattern matching scheme using the '*' character. For example, 131.120.10.* would indicate the inclusion of all hosts with IP *addresses* between 131.120.10.0 and 131.120.10.255.

When comparing two *addresses* in PPL for overlap, there has to be an exact match to the end of the *address*, or until the first '*' is reached. For example 131.120.*.* would include all machines specified by the address 131.120.10.*, and therefore an overlap would exist.

The first evaluation of *address* overlap was the pattern matching scheme just discussed. The second evaluation for overlap of address space is based on Table 5. This evaluation takes into account the operators that can be used in specifying *address* ranges, namely == and !=. In this table, *addresses* are represented by either an X or Y. When the letters match between the two columns representing *Address A* and *Address B*, this indicates the *addresses* themselves include at least one host in common. With the example stated earlier, one policy rule specified 131.120.*.* and the second rule specified 131.120.10.*, would result in an overlap of all *addresses* beginning with 131.120.10.*. Since the *addresses* have a range in common the same letter would appear in both columns. If the two addresses being compared were rather 131.*.*.* and 141.*.*.*, then according to PPL there would be no overlap in *address* space and a lookup into the table would only consider those columns with both an X and Y.

Address A	Address B	Result of Combined Address Resolution
= X	= X	Overlap
= X	= Y	No Overlap
= X	!= X	Conflict
= X	!= Y	Overlap
!= X	!= X	Overlap
!= X	!= Y	Overlap
-	= Y	Overlap
= X	-	Overlap
-	-	Overlap

Table 5. Conditional Address Overlap

In general, the results of comparisons in Table 5 are similar to other *conditional* attributes discussed in this chapter. The result can be any of three values, *conflict*, *overlap*, or *no overlap*. Because multiple *addresses* can be specified, each address grouping has to be compared. If any of the comparisons result in a conflict, then there is no hope of overlap and no need for continued overlap checking. When a *no overlap* is returned, this only indicates that although there is no overlap between two particular address groupings, there is still a potential for overlap with other addresses specified in the *conditional* section of the rule. In this case, if no *overlap* status is returned, then the result is *no overlap* and there is no chance for a conflict between the two policy rules. When an *overlap* status is returned and no *conflict* is returned, the potential for policy conflict does exist, and further evaluation is necessary. When an *address* specification is not included in a policy rule, the default assumption is all *addresses* are effected by the policy rule, and therefore an overlap exists. No *address* specification is represented in the table by the '-' character.

To provide support for future expansion, PPL allows user defined types to be included in the *conditional* section of a policy rule. This *type* feature allows for further granularity of policy specification. In the following tables a *type* of day will be used and defined as:

define type day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

Day is defined to be a set containing seven elements which represent the days of the week. An order is implied with these elements, so the use of operators ' \leq ' and ' \geq ' as well as ' $=$ ' and ' \neq ' are allowed. When multiple days are to be associated with a policy, then each will be specified as needed. For example, if a user wanted to limit traffic to Wednesday and Thursday only, the following condition list could be given: $\{day == Wed, day == Thu\}$. If a policy was only needed during the normal work week then the following condition list could be used: $\{day \neq Sat, day \neq Sun\}$.

Table 6 represents the expansion that takes place during the conflict checking phase of the compilation. This expansion is strait forward, if a day is associated with the equality operator then that day is added to the expanded list. When a non equality operator is used, then that day is taken away from the expanded list. If multiple days are denied in the conditional section of the policy rule, then an intersection of the possible days is made with each day denied, see third row of Table 6. The results of the expansion will always be a list of elements were the ' $=$ ' symbol is

assumed to apply. For example, when the policy specifies $\{day \leq Wed\}$, the result will be $\{day == Sun, day == Mon, day == Tue, day == Wed\}$.

Expressed Type	Expanded Type
$day == Sat.$	Sat.
$day != Sat.$	Sun, Mon, Tue, Wed, Thu, Fri
$day != Sat, day != Sun$	Mon, Tue, Wed, Thu, Fri
$day \leq Wed$	Sun, Mon, Tue, Wed,

Table 6. Conditional Expansion of Type

Table 7 represents the logic used in the determination of overlap in an expansion of a user defined *type* statement. As mentioned earlier the only operator that needs to be considered after a *type* expansion is the '=' operator. As a result, the only time an overlap condition will exist for a user defined *type*, is when the values specified contain at least comparison were the values are equal.

Type A		Type B		Result of Combined Type Resolution
=	X	=	Y	No Overlap
=	X	=	X	Overlap
-	-	=	Any Value	Overlap
-	-	-	-	Overlap

Table 7. Conditional Type Overlap

Priority can be assigned to network traffic as well in PPL policy rules. The use of *priority* can give preferential treatment to certain types of traffic. For example, voice transmission over the network could be given preferential treatment over normal data traffic because of the QoS demands needed to support voice transmission. Table 8 shows the logic used to compare two *priority* values for an overlapping condition. The operators '<=' and '>=' are the only allowed in specifying a *priority* conditional attribute. *Priority* values are numeric and

therefore can be ordered for comparison with these operators. An exact *priority* can be issued with the combination of both an ' \leq ' and ' \geq ' operators. For example, when network traffic is allowed only with a *priority* of 10, this can be represented with the following conditional statement: {*priority* \geq 10, *priority* \leq 10}. Conjunction is used with all conditional statements therefore all the statements have to evaluate to true for an overlap. In Table 1 an '*' symbol in the operator column allows either ' \leq ' or ' \geq ' symbols to be used. A '*' symbol in the *priority* value, field implies any numeric value can be used. To represent the same numeric value an 'X' symbol is used in the *priority* value field. Because of the equality in both *priority* operators, whenever the value of the *priorities* being compared are equal, there is an overlap. The numbers 1 and 2 are used to represent that fact that one value is greater than another. In reality these values can be any numbers that preserve this feature. When an '-' symbol is used in the table below it represents the fact that no *priority* attribute was explicitly represented in a policy rule. The assumption is that when no *priorities* are stated, then all *priorities* are allowed.

Priority A		Priority B		Result of Combined Priority Resolution
\geq	*	\geq	*	Overlap
\leq	*	\leq	*	Overlap
*	X	*	X	Overlap
\geq	1	\leq	2	Overlap
\geq	2	\leq	1	No Overlap
\leq	2	\geq	1	Overlap
\leq	1	\geq	2	No Overlap
-	-	*	*	Overlap
*	*	-	-	Overlap
-	-	-	-	Overlap

Table 8. Conditional Priority Overlap

To prevent data packets from looping for infinity through a network, a *hop count* can be associated with a packet which represents the packet's maximum lifetime. PPL enables a *hopcount* attribute to be used in the *conditional* section of a policy rule to support the limiting of

a packets lifetime. This *hop count* is decremented each time the packet passes through a router. When the *hop count* reaches zero, the packet is dropped from the network. Using the operators '<=' and '>=', two policy rules can be checked for overlapping conditions. Table 9 represents the logical relationship used by the PPL compiler to determine if an overlapping condition exists, and therefore the possibility for conflict. In the table, an '*' symbol represents any numeric value when placed in the *hopcount* value field, and either of the *hop count* conditional operators when placed in the operator field. An '-' symbol represents that fact that no *hopcount* attribute was specified for policy rule and therefore will support all values. The numbers 1 and 2 are again used represent that the caparison is being made between two values where one is possible greater than the other. An 'X' symbol is used to represent the fact that two *hopcounts* are exactly the same.

Hopcount A		Hopcount B		Result of Combined Hopcount Resolution
>=	*	>=	*	Overlap
<=	*	<=	*	Overlap
*	X	*	X	Overlap
>=	1	<=	2	Overlap
>=	2	<=	1	No Overlap
<=	2	>=	1	Overlap
<=	1	>=	2	No Overlap
-	-	*	*	Overlap
*	*	-	-	Overlap
-	-	-	-	Overlap

Table 9. Conditional Hopcount Overlap

Bandwidth is the term used in PPL to represent the data rate of traffic in units/sec. The units can vary from gigabits, megabits, kilobits, and bits. During the compilation all units are converted to megabits, to provide a common base for comparison. *Bandwidth* is used to determine conflicts in two possible ways. During the establishment of links in a PPL policy file, each link is specified with a supporting bandwidth. When no bandwidth is specified, a zero

bandwidth is assumed. The first check of conflicting bandwidths is during the establishment of paths through the network. Each network link along a longer path must be able to support the total bandwidth requirement of that longer path.

The second use of *bandwidth* for detecting conflicts is in the determination of overlap in the *conditional* section of the policy rule. During the earlier conflict detection process the operator '=' is exclusively used to represent the *bandwidth* parameters. In the *conditional* section the operators '<=' and '>=' are used for comparison. These *bandwidth* requirements are used to check for minimal *bandwidth* performance levels. An example of two *bandwidth* statements being compared can be seen in the following two PPL policy rules.

```
Policy_bandwidth_1 net_manager {*} {traffic_class = {data}} {bandwidth >= 20 MBPS}
{deny}
```

```
Policy_bandwidth_2 net_manager {*} {traffic_class = {data}} {bandwidth >= 40 MBPS}
{permit}
```

In this case each policy is specifying data network traffic *bandwidth* requirements that are above 20 and 40 Mbps respectively. The *bandwidth* comparison made during the conditional phase of the compiler would determine that there is an overlapping *bandwidth* requirement, namely when the *bandwidth* requirement is above 40 Mbps. This overlapping condition will lead to the detection of conflicting policy rules where one rule denies traffic above 40 Mbps and the other permits it. Table 10 represents the logic used in the PPL compiler to determine if an overlap of *bandwidth* requirements exist. As with the other tables in this chapter the '*' symbol represents any valid value for that column, and the '-' symbol represents that fact that no *bandwidth* attribute was specified in the *conditional* element of the policy rule. The 'X' symbol is used to relate the fact that both values are equal and the numbers 1 and 2 are used to indicate that one value is greater than the other.

Bandwidth A		Bandwidth B		Result of Combined Bandwidth Resolution
>=	*	>=	*	Overlap
<=	*	<=	*	Overlap
-	X	-	X	Overlap
>=	1	<=	2	Overlap
>=	2	<=	1	No Overlap
<=	2	>=	1	Overlap
<=	1	>=	2	No Overlap
-	-	*	*	Overlap
*	*	-	-	Overlap
-	-	-	-	Overlap

Table 10. Conditional Bandwidth Overlap

4. Find Overlapping Paths using Expanded *Path* and *Conditional* Elements

The process of finding overlapping *paths* is straight forward yet tedious. After the expansion of any *path* containing wildcard characters, each *path* is compared against one another other for an overlapping *path segment*. That overlapping *path segment* could be either one node or one link between the two *paths*. When a common *path segment* is found, the *conditional* elements of each policy are compared using Table 3 though Table 10 to determine if the overlapping *path segments* can co-exist because of further granularity; see Section 3 for details. If the paths are not from the same policy rule, using the *policyID* to determine this, then two policy rules have a potential for conflict and are marked for the verification of conflict using the *target* and *action items* elements, details are described below.

5. Expanding Target Elements with Consideration of Action Items

To help detect conflicting policies, the *target* element does not support implicit permits. An *action* element value of 'permit' allows the situation described by the combination of *targets*, *conditions*, and *paths*, to exist. The use of implicit policies can lead to ambiguity in expressing

network policies. If there are to be implicit actions, then the action should be on the side of denial rather than permission. PPL supports implicit denies, but not implicit permits. As a result, when a user wants traffic to be permitted on a *path*, they must explicitly state this fact, or permit all traffic with the '*' symbol as the only target attribute. For example, a class called "traffic_class" is defined below.

define class traffic_class {data, video, voice};

This class is a set that contains three values: *data*, *video*, and *voice*. Table 11 shows the logical results enforced by the PPL compiler with the combination of *target* information and *action items* using the defined traffic_class.

Target	Action	Result
traffic_class = {voice}	permit	explicit permit: voice implicit deny: data, video
traffic_class = {voice}	deny	explicit deny: voice implicit deny: data, video
traffic_class != {voice}	permit	explicit permit: data, video implicit deny: voice
traffic_class != {voice}	deny	double negative: There is no explicit permit. explicit deny: data, video implicit deny: voice
*	permit	Explicit permit on all classes of defined traffic, all elements of the set
*	deny	Explicit deny of all targets explicit deny: data, video, voice

Table 11. Target and Action Combination

The results of the combination of *target* elements and actions are associated with the *policy ID* from the policy rule. When overlapping *paths* are detected between two policy rules, the results of the expansion of the *target* elements are used to determine if a conflict exists.

The *action item* can contain more than just the two values, *permit* and *deny*. The other *action items* are to control the dynamic aspects of the network. The keywords *priority* and

hopcount can be used to change values associated with the keywords when the network reaches the state which is described by the combination of *path*, *target*, and *conditions*. For example, if the loss rate of voice data reaches a certain level, the *priority* of this voice traffic may be increased in an attempt to provide better QoS to users using voice over the Internet. When the keywords *hopcount* or *priority* are used in the *action items* element, the implicit action is 'permit'. When the *action item* value is 'deny', then no other action item elements can exist in the list. The compiler will catch this type of conflict, both permit and deny within the same policy rule, and flag it as a syntax error. The following policy rule explicitly denies the *data* traffic_class on the NPS path.

```
policy5 net_manager {NPS} {traffic_class == {data} *} {DENY};
```

It is possible to have multiple permit *actions*, such as changing the *priority* of the traffic described by the policy rule as well as changing the *hopcount* associated with the policy rule.

C. RESOLVING CONFLICTS

The PPL compiler supports the automatic resolution of conflicting policies with the use of *policy ID*. The *policy ID*, is the element of the policy rule that identifies the creator of the rule. The *policy ID* field maps to an integer value, which permits the comparison of two *policy ID*'s to determine if one is greater than the other. When two policy rules have the same creator, or when each of the creators has the same priority level, automatic conflict resolution may not be possible. By default the conflict tester will not resolve conflicts between policies created by the same user or precedence level. As will be covered later in this chapter, a class of traffic not explicitly permitted, is implicitly denied in PPL. There may exist a situation when a single user creates several policy rules to create an overall network policy. Rather than explicitly listing all classes of effected traffic in each incremental policy rule, a compiler option may be used to alleviate this inconvenience. This option informs the policy tester to ignore conflicts that exist because of implicit denies between policy rules created by the same user. For example, take the following two policies:

```
Policy1 net_manager {Path1} {class == {faculty}}{*}{permit};
```

```
Policy2 net_manager {Path1} {class == {student}}{*}{deny};
```

With the default option of the compiler, these two policy rules would produce a conflict, since *faculty* traffic is permitted by *Policy1* on *Path1*, but implicitly denied by *Policy2*. Using the “no_implicit_deny” option during compilation, this conflict will be ignored since it was produced as a result of an implicit deny between two policies created by the same user.

When a conflict does exist between two policy rules, and the ID values are not equal, then the higher value of the two ID's will have priority and in a sense deactivate the lower priority policy rule. The PPL compiler will advise the user which policy rules were resolved and how. The *policy ID* will allow a user to implement their own local policy rules, but when in conflict with a higher ID, the network administrator for example, the local policy rule will be overridden.

In its current state, the PPL compiler accepts a configuration file that is created with an ordinary text editor. There is no control over its creation or the information entered into it. A mechanism must be created that prevents unauthorized access to this file, or the misrepresentation of network policy creators. One means to accomplish this would be to establish an interface that prevents direct access to the configuration file. This interface would allow policies to be created by password protected accounts, were only the current user's identifier would be associated with a policy rule. Accounts to the interface would be established that assign a priority level to each user. The priority levels are not required to be unique, but when multiple users are assigned the same priority level, there is a potential for an unresolved conflict to exist.

Although the current PPL compiler only supports *policy ID* for conflict resolution, in theory there is no reason that further means cannot be implemented in the future.

D. PERFORMANCE

Performance was not emphasized during the implementation of the compiler and policy tester. More important was the ability to correctly detect conflicts between network policies. The detection of network policy conflicts is envisioned as an offline process that is executed before the policies are actually implemented in the network. Basing network size on a typical autonomous system of 40 nodes, network simulations were run. The results of three simulated networks are provided in Table 12 providing time estimates to process and test for conflicting policies. The three networks varied the number of nodes comprising each network at 10, 30, and

40. Analysis of the results confirmed that performance is directly related to the number of network paths that need to be generated for wildcard matching. The flexibility of the path representation provided by PPL, is also a performance bottleneck.

PPL provides the use of wildcards in representing network paths to support aggregation. When a policy must be applied to all possible paths from node 1 to node 20 for example, it can be represented by <Node1, *, Node20>. Rather than listing all the paths explicitly, which in some cases may be hundreds, this representation allows one statement to represent them all. Path aggregation allows for smaller policy databases, as well as requiring less bandwidth to transmit path information to network enforcement points. The down side, is that the policy tester will generate all possible paths through the network as part of the path expansion process.

To improve time performance, a flag can be used to inform the compiler that all policies represented in the data file explicitly list each node, and that no path expansion is required. Table 12 displays the time required and the number of paths generated both with, and without the compilation flag. The network consisting of 10 nodes includes paths the contain wildcard characters, and therefore could not use the "no wildcard" flag.

Number of Nodes	No. Policies	Wildcards		NO Wildcards	
		No. Paths	Time (min)	No. Paths	Time (min)
10	11	377	0:15	-	-
30	10	1449	6:09	0:09	37
40	20	2271	18:35	0:09	126

Table 12. Simulation Results

E. CHAPTER SUMMARY

A major contribution made by PPL is the ability to test for conflicting network policies before they are disseminated throughout the network. This chapter discusses the formal logic representation of the network policies, and the incremental steps taken to detect conflicts using

the Prolog Interpreter. Tables are presented throughout this chapter to state the logical consequence of comparison operations between policy rules. The chapter is concluded with a discussion of the method used for resolving conflicts by using the identifier of the policy's creator.

V. CASE STUDY

As a proof of concept, a case is presented in which network policies, represented in PPL, are tested using formal methods to determine the consistency of those policies. To demonstrate this procedure, network policies are presented which apply to the network represented by Figure 15.

A. A 10 NODE SIMULATED NETWORK

Suppose that ten organizations, which are represented in Figure 15, are connected together as represented by this diagram. These organizations wish to apply policies to control

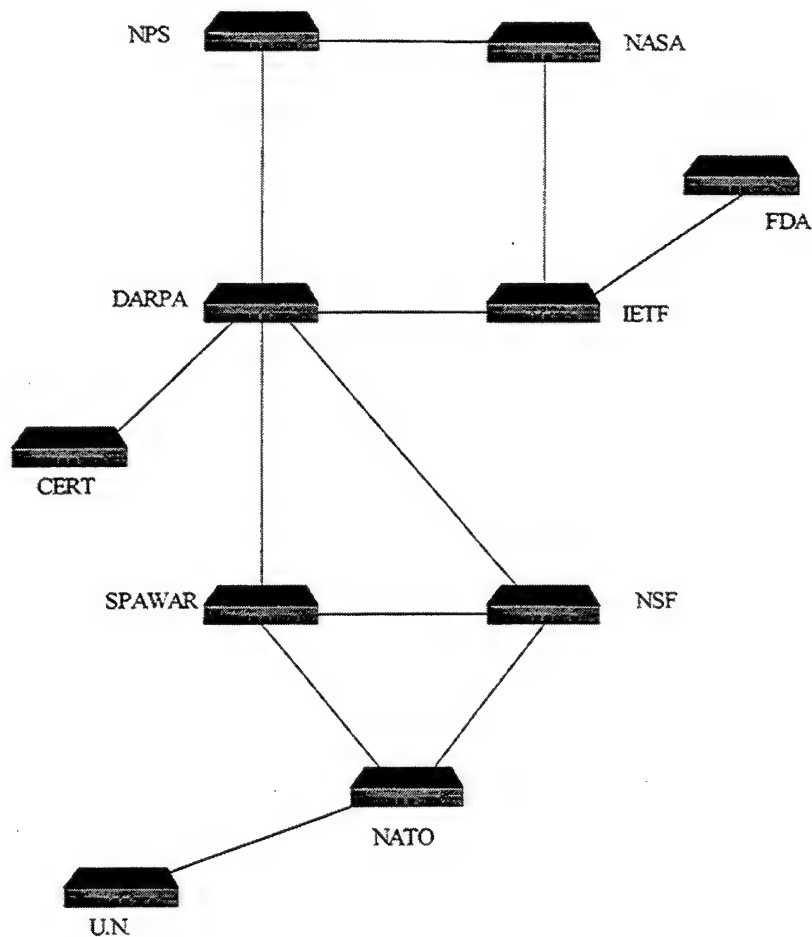


Figure 15. Case Study Network Diagram

the performance and access to their network. Using the Path-based Policy Language (PPL), network's connectivity is defined using nodes and links and entered into a configuration file. This configuration file also contains user defined classes of traffic, data types, parameters applied to the nodes and links, as well as the network policies and the paths they apply to. The full configuration file can be found in appendix B. In the next few sections the elements of the configuration file will be discussed and explained. Although small, perhaps ¼ the number of nodes associated with a typical Autonomous System, this network will allow us to demonstrate the features of PPL and the ability for conflict detection between policies. The benefit of a smaller network diagram, is the ability of not overwhelming the reader with too much complexity.

B. NETWORK MODELING AND USER DEFINED DATA

The first step in modeling a network is to define the nodes available for creating the links and paths for which network policies will be applied. This is done with simple define statement identifying a name with a node in a network. The nodes in this network reflect real world organizations, but the connectivity of these organizations in the real world is not implied. The nodes can be defined with one or more statements as a comma separated list of names. The following statements define the 10 nodes that comprise the network.

```
define node NPS, DARPA, SPAWAR, NSF, IETF, NASA;
```

```
define node FDA, NATO, UN, CERT;
```

The connectivity of the network is represented by defining the links between nodes. Although the link define statements imply a direction associated with the link, the PPL compiler and conflict detection process ignore direction and apply bi-directional communication. The following list of define statements represent the eleven links that create the network. The ability to list several links with the same define statement is a time and space saver that again can be used with the definition of links.

```
define link NPS_DARPA <NPS, DARPA>, NPS_NASA <NPS, NASA>;
```

```
define link DARPA_IETF <DARPA, IETF>, DARPA_NSF <DARPA, NSF>;
```

```
define link DARPA_SPAWAR <DARPA, SPAWAR>;
```

```
define link DARPA_CERT <DARPA, CERT>;
```

```
define link NASA_IETF <NASA, IETF>;
```

```

define link IETF_FDA <IETF, FDA>;
define link SPAWAR_NSF <SPAWAR, NSF>;
define link SPAWAR_NATO <SPAWAR, NATO>;
define link NSF_NATO <NSF, NATO>;
define link UN_NATO <UN, NATO>;

```

All paths must be defined before they can be referenced by a PPL policy rule. This definition of paths, allows for attributes to be associated with those paths. These attributes include the bandwidth requirement for the path, and the messages required to support policies based on dynamic features of the network such as delay, jitter, and data loss. A path can be represented by an explicit listing of nodes required to construct the path, or may also include the wild card character '*' which expands to match all possible paths. The following define statements were used in the configuration file to define the paths and the attributes associated with the paths, links and nodes that comprise the network.

```

define path NPS_CERT {<NPS, *, CERT>;};
define path NPS_NSF {<NPS, DARPA, NSF>;};
define path NASA_SPAWAR {<NASA,IETF,DARPA,SPAWAR>;};
define path UN_NPS {<UN,*,NPS>;};
define path ALL {*};
define link_param NPS_DARPA {BW := 100MBPS, delay(), loss_rate()};
define path_param NPS_CERT {BW := 100 MBPS, jitter()};
define path_param NPS_NSF {BW := 100 MBPS, jitter()};

```

The path statements defined vary from the very specific, listing every node and the order which it will appear, to the path defined as *ALL*, which represents all possible paths in the network. The *ALL* path, is useful when you want to apply a single policy over the entire network. The path labeled *NPS_CERT* represented as <NPS, *, CERT>, is used to identified all valid paths in the network that have a source of NPS and a destination of CERT. The use of the term valid path, implies the path are loop-free.

Parameters are associated with the links and paths using the appropriate define statement, either *link_param*, or *path_param*. The parameters associated with the link *NPS_DARPA* are that a *bandwidth* of 100 Mbps is supported, as well the *messages delay()*, and *loss_rate()*. As mentioned earlier, these *messages* support dynamic policies where periodic input

about the state of the network is required. These *messages* are simple identified, not defined in any rigid sense. The idea is that the policy has no need to know what information is passed in these *messages* or the format of that information. This allows network policies to utilize user defined *messages*, which required no changes to the formal PPL grammar. Although there must be an agreement between the policy server and other devices on exactly what information is to be carried by the *message* and the format.

Other than the policies themselves, the remaining information that is entered into the configuration file are the user defined elements which will be used to construct the network policies. Every policy has a creator, so all the valid users who may create policies must be listed with their associated priority. The priority is used during the conflict resolution phase to negate a policy when created by a user with a lower priority. The highest priority is 1, and in this case only the network manager is assigned this value. The valid users and their priorities for this network scenario are listed below.

```
define policy_maker Net_Manager(1), Xie(3), Lundy(3), Stone(4);
```

Traffic types that are used throughout the network policies are listed below. The user may add, delete, or modify these classes to support their own needs. In this case defined six classes of traffic. Each traffic class contains all the valid values that may be used when referring to this class.

```
define class traffic_type {research, university};  
define class traffic_class {data, video, voice};  
define class traffic_security {Private, Public};  
define class traffic_priority {High, Med, Low};  
define class user {faculty, student, staff, accounting};  
define class node_traffic {NPS, DARPA, SPAWAR, NSF, IETF, NASA, FDA,  
NATO, UN, CERT};
```

The user may also define *types* as well. Types are similar to the user defined classes in that they are assigned a label and are associated with a list of valid values. Classes are used in the *target element* of a policy rule, and *types* are used in the *conditional element* of a policy. There is an ordering associated with the values of a *type*, so the comparison operators '<=', '>=', '==', and '!= ' may all be used. A *type* defined and used in the network policies is listed below. This *type* is used to represent the seven days of the week.

define type day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

C. NETWORK POLICIES

The policies which are applied to the network represented by Figure 15, are also entered into the configuration file and described below.

Policy1 was created by the network manager and applies to the *path* *NPS*→*DARPA*→*NSF*. This policy permits *research* traffic to and from the nodes *NPS* and *NSF*. A further restriction is placed on this traffic to only allow it between the hours of 0800 and 1200, Monday through Friday.

**Policy1 Net_Manager {NPS_NSF}
 {traffic_type = {research}, node_traffic = {NPS, NSF}}
 {day != Saturday, day != Sunday, time >= 0800, time <= 1200}
 {permit};**

Policy2 was created by the user *Stone* and applies to the *path* *NASA*→*IETF*→*DARPA*→*SPAWAR*. The user *Stone* has a priority level of 4, which is the lowest of all defined users. This policy denies traffic from the *NSF* to flow along this *path*.

Policy2 Stone {NASA_SPAWAR} {node_traffic = {NSF}} {*} {deny};

Policy3 was created by the user *Lundy* and applies to all *paths* between the nodes *NPS* and *CERT*. The wildcard character was used in the definition of this *path* and when expanded represents two *paths* through the network. The first is *NPS*→*DARPA*→*CERT*, and the second is *NPS*→*NASA*→*IETF*→*DARPA*→*CERT*. Although there are other possible paths between *NPS* and *CERT*, these two are the only ones that are free of loops. Loops in a network are not desirable, and are not considered valid during the expansion process. This policy permits all traffic without restriction to flow over this path.

Policy3 Lundy {NPS_CERT} {*} {*} {permit};

Policy4 was created by the *network manager* and applies to link between the nodes *NASA* and *IETF*. This policy permits all traffic without restriction to flow on this link.

Policy4 Net_Manager {NASA_IETF} {*} {*} {permit};

Policy5 was created by the *network manager* and applies to link between the nodes *NPS* and *DARPA*. This policy denies *university* traffic from passing over this link after 1100.

```
Policy5 Net_Manager {NPS_DARPA}
    {traffic_type == {university}}
    {time >= 1100}
    {deny};
```

Policy6 was created by the *network manager* and applies to the same link as Policy5, the link between the nodes *NPS* and *DARPA*. This policy denies *university* traffic from passing over this link after 1300 each day.

```
Policy6 Net_Manager {NPS_DARPA}
    {traffic_type == {university}}
    {time >= 1300}
    {deny};
```

Policy7 applies to the node *NSF*. As mentioned earlier, the ability to specify policies that apply to nodes, provides the ability to scale well in larger networks. In this case *NSF* represents a sub-network that denies all traffic with a host address starting with 131.

```
Policy7 Net_Manager {NSF} {*} {hostIP == 131.*.*} {deny};
```

Policy8 applies to multiple *paths* as did Policy3, in this case when the *path* is expanded as a result of the wildcard used in it's declaration, eight paths are effected. These eight unique paths permit all traffic which match the host addresses of 153.20.8 or 131.40.*.* to flow through the paths.

```
Policy8 Xie {UN_NPS} {*} {hostIP == 153.20.8.*, hostIP == 131.40.*.*} {permit};
```

Policy9 applies to the link connecting the nodes *NPS* and *DARPA*. Before the hour of 0800, *video traffic* is permitted on the link and is assigned a *priority* of 3.

```
Policy9 Net_Manager {NPS_DARPA} {traffic_class == {video}}
    {time <= 0759:59} {priority := 3};
```

Policy10 also applies to the link connecting the nodes *NPS* and *DARPA*. This policy denies *video* traffic between hours of 0800 and 1600 everyday.

```
Policy10 Net_Manager {NPS_DARPA} {traffic_class == {video}}
    {time >= 0800, time <= 1600} {deny};
```

Policy11 again applies to the link connecting the nodes *NPS* and *DARPA*. This policy permits *video* traffic after the hour 1600, but lowers the *priority* to 5.

```
Policy11 Net_Manager {NPS_DARPA} {traffic_class == {video}}
    {time >= 1601} {priority := 5};
```

D. DETECTED CONFLICTS

Once the information required to model the network and represent the network policies is entered into the configuration file, the next step is to compile the file. This compilation phase parses the configuration data, including the network policies, to validate its representation is correct. The output of the compilation is automatically fed into a theorem prover based on formal logic. In this case the theorem prover is a program written in Prolog that defines a conflict, and along with the data extracted from the configuration file, is used to verify policy consistency. The results of the compilation and theorem proving phase are described below. There were a total of 7 conflicts that could *not* be resolved, and 22 conflicts which were resolved. The creator of the policy was used as a means of precedence to nullify policies of lower priority users, and therefore resolve certain conflicts. The complete results of the conflict detection process are listed in appendix C. The number of conflicts both resolved and not resolved can be a little misleading. The fact that one policy may be applied to multiple paths at the same time requires each path to be verified individually. For example, suppose that there are two policies defined for a network, policy1 and policy2. Policy1 applies to two paths through the network, but policy2 only effects one link of network. During the conflict detection phase, one of the paths associated with policy1 was determined to conflict with the single link of policy2, but the second path of policy1 did not conflict with the link of policy2. For completeness, every path effected by the conflict is listed separately. This allows the user to identify which path of the network is effected. But when both paths of policy1 are in conflict with the single link of policy2, two conflicts are listed. This may be a little misleading, in that the conflict between policy1 and policy 2 is listed twice, once for each path. This is exactly what happened in this case study. There were 7 conflicts listed which could not be resolved, but there were actually only 5 conflicts between policy rule pairings. The 22 non-resolved conflicts were reduced to only 8 unique policy rule pairings. The 13 policy conflicts are represented in Table 13. Each of these 13 conflicts are addressed individually below.

Unresolved Policy Conflicts		
1	Policy 10	Policy 1
2	Policy 5	Policy 11
3	Policy 5	Policy 1
4	Policy 6	Policy 11
5	Policy 7	Policy 1
Resolved Policy Conflicts		
6	Policy 10	Policy 8
7	Policy 10	Policy 3
8	Policy 4	Policy 2
9	Policy 5	Policy 8
10	Policy 5	Policy 3
11	Policy 6	Policy 8
12	Policy 6	Policy 3
13	Policy 7	Policy 8

Table 13. Policy Conflicts

Conflict 1: Policy 10 specifies that *video* traffic is denied between the hours of 0800 and 1600 on the link between *NPS* and *DARPA*. Policy 1 applies to a *path* between *NPS* and *NSF*, specifically *NPS*→*DARPA*→*NSF*. Both these policies share the link *NPS*→*DARPA*, but Policy 1 permits *video* traffic between 0800 and 1200 on all days except *Saturday* and *Sunday*. The conflict process starts with the fact that two policies both permit and deny the same traffic class on a common link. To verify the conflict, the *conditional* elements were compared for overlap. The fact that the *conditions* of Policy 10 did not mention which *days* applied to the policy, implied that all *days* were covered. The result was an overlap of the *days Monday, Tuesday, Wednesday, Thursday, and Friday*. The remaining *conditional* element of these two policies was the *time* specified. When compared, there was also an overlap of the hours between 0800 and 1200. This information confirmed that a conflict existed, but the creators of these two policies may be used to resolve the conflict by nullifying the policy created by the lower priority user. In this case both policies were created by the *network manager*, and thus can not be resolved.

Conflict 2: Policy 11 and Policy 5 both apply to the same link, *NPS_DARPA*. Policy 11 assigns a priority to *video* traffic after 1600 each day. When an action such as assigning a *priority* is used, there is a default action of permit also implied. Policy 5 makes no reference to *video* traffic and therefore a denial of this traffic is implied. Unless a traffic class is explicitly permitted in a policy rule, the default action is to deny it. Policy 11 permits *video* traffic after 1600 each day and Policy 5 denies *video* traffic after 1100 each day. The *hours conditions* from these two policies form an overlap anytime after 1600. This causes a policy conflict, and since each policy was created by the *network manager*, no resolution is possible with the default options. Using the “-no_implicit_deny” option discussed in section C titled “Resolving Conflicts”, this conflict would be avoided because it was caused by an implicit deny.

The fact that Policy 5 explicitly denies *university* traffic has no effect on the conflicting situation, since the *university* traffic is not addressed in Policy 11, it is implicitly denied.

Conflict3: The third conflict identified was between Policy 5 and Policy 1. Policy 1 applies to the *path NPS→DARPA→NSF*, and Policy 5 applies to the link *NPS→DARPA*. These two *paths* overlap on the *NPS* to *DARPA* link. Policy 1 permitted *research* class traffic as well as *NPS* and *NSF* traffic on this *path* during non-working hours. Policy 5 explicitly denies *university* class traffic after the *hour* of 1100, and since no mention of *research*, *NSP* or *NSF* is made in the target element of the policy rule, they are implicitly denied. These two policies conflict in the fact that *research*, *NPS*, and *NSF* is permitted in Policy 1, and denied in Policy 5. The *conditional* element of the policy rules overlap between the hours of 1100 and 1200, *Monday* through *Friday*. This conflict can not be resolved using default options again, since both were created by the *network manager*. As mentioned in Conflict 2, the “-no_implicit_deny” option would also have resolved this conflict.

Conflict 4: Policy 6 and Policy 11 are both applied to the *NPS→DARPA* link of the network. Policy 11 permits *video* traffic on the link after the hour of 1600 each day. Policy 6 makes no mention of *video* traffic in its policy rule and therefore an implicit deny of *video* traffic is applied. Policy 6 is enforced after the hour of 1300 each day, leaving an overlap of hours between 1300 and 1600 each day. Without the use of the “-no_implicit_deny” option, this conflict can not be resolved by the policy’s creator since the *network manager* was the creator in both policies.

Conflict 5: The last conflict that can not be resolved using the policy's creator, was between Policy 1 and Policy 7. Policy 1 applied to the *path* $NPS \rightarrow DARPA \rightarrow NSF$, and Policy 7 applied to the *NSF* node itself. When policies are applied to a node, this indicates that the node represents a subnet itself. The overlap between these two *paths* is obviously the *NSF* node. Policy 1 permits *research*, *NPS* and *NSF* traffic during certain times of the day. Policy 7 denies *university* traffic and with no mention of *research*, *NPS*, or *NSF* traffic, these are implicitly denied as well. The denial of *research*, *NPS*, and *NSF* traffic of Policy 7 is applied to any network address beginning with 131. The support of *addresses* beginning with 131 is implied in Policy 1, therefore a contradiction occurs with regards to the common traffic classes. Again, both policies were created by the same user, causing no automatic resolution to be possible.

Conflict 6: This conflict is between Policy 8 and Policy 10. Policy 8 applies to a *path* that utilizes the wild card character '*' in its definition. The path is defined as $CERT \rightarrow '*' \rightarrow NPS$ and includes all possible *paths* between the nodes *CERT* and *NPS*, which do not include loops. This wild card matching provided in this definition provides eight unique *paths* which are listed below.

$UN \rightarrow NATO \rightarrow NSF \rightarrow \mathbf{DARPA} \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow SPAWAR \rightarrow \mathbf{DARPA} \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow NSF \rightarrow SPAWAR \rightarrow \mathbf{DARPA} \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow SPAWAR \rightarrow NSF \rightarrow \mathbf{DARPA} \rightarrow NPS$

$UN \rightarrow NATO \rightarrow NSF \rightarrow DARPA \rightarrow IETF \rightarrow NASA \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow SPAWAR \rightarrow DARPA \rightarrow IETF \rightarrow NASA \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow NSF \rightarrow SPAWAR \rightarrow DARPA \rightarrow IETF \rightarrow NASA \rightarrow NPS$
 $UN \rightarrow NATO \rightarrow SPAWAR \rightarrow NSF \rightarrow DARPA \rightarrow IETF \rightarrow NASA \rightarrow NPS$

The first four paths include an overlapping link, $DARPA \rightarrow NPS$, with Policy 10. The last four paths of Policy 8 do not overlap with Policy 10, and therefore can not cause a conflict. The results of the conflict detection phase identified four conflicts between these policies as a result of the four unique *paths* which overlapped with Policy 10. Solid links in Figure 16 show the links that are covered by the eight expanded possible *paths* of Policy 8. The links not effected by Policy 8 are indicated with dashed lines. As a result of this coverage, Policy 8 has a great change of overlapping with other policies, and therefore may result in a greater number of conflicts.

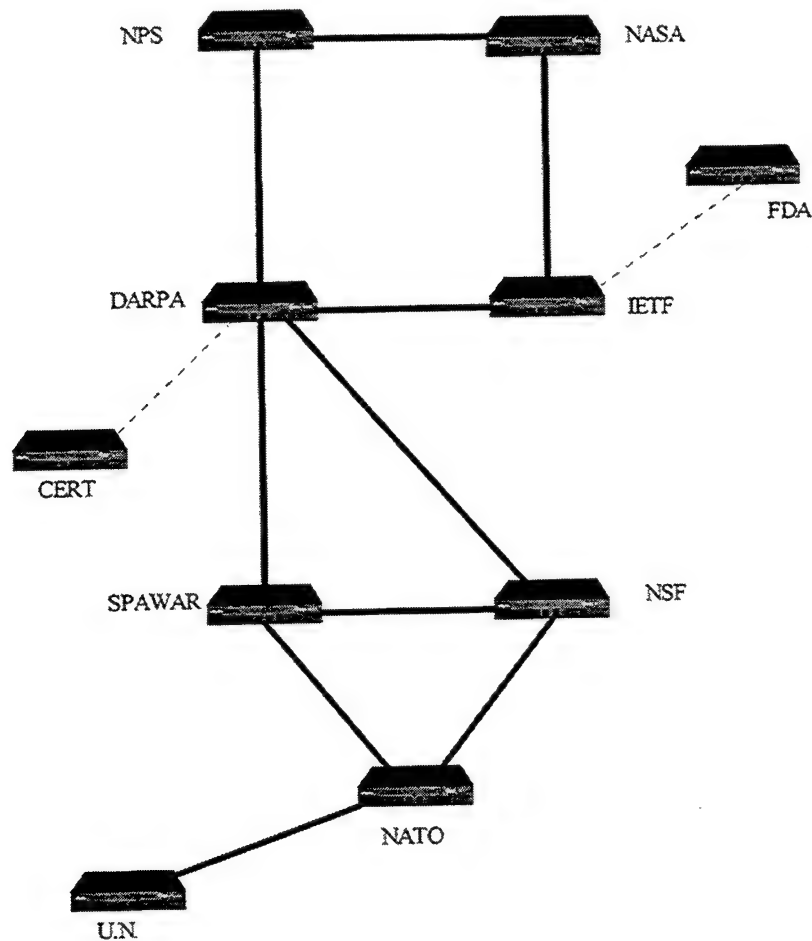


Figure 16. Link Coverage from Policy 8

Policy 8 permits all traffic classes that have a source or destination *address* beginning with 153.20.8 or 131.40 to flow over that *path*. Policy 10 denies *video* traffic on the *DARPA*→*NPS* link between the *hours* of 0800 and 1600 regardless of the *address*. These two policies create a conflict were *video* traffic is denied, by Policy 10, and permitted by Policy 8. There is a time overlap of the *hours* 0800 through 1600, and as a result of not specifying *addresses* in the *conditional* element of Policy 10, all *addresses* are permitted, causing a conflict between the policies.

Policy 8 was created by the user *Xie* and has precedence of 3. Policy 10 was created by the *network manager* and has a precedence level of 1. As a result of these differing precedence, the conflict can be resolved by nullifying the policy enforcement of the 4 overlapping paths

specified by Policy 8. The four remaining *paths* that did not include the *DARPA*→*NPS* link will be able to support Policy 8.

Conflict 7: This conflict is between Policy 3 and Policy 10. Policy 3 also utilizes a *path* that was defined with the wild card character '*', namely *NPS*→*→*CERT*. When expanded, this path produces two unique *paths* through the network that are loop free. These *paths* are *NPS*→*DARPA*→*CERT* and *NPS*→*NASA*→*IETF*→*DARPA*→*CERT*. The first *path* overlaps with link that Policy 10 applies to, *NPS*→*DARPA*.

Policy 3 permits *all traffic* to flow with no *conditions*. Policy 10 denies *video traffic* during certain times of the day. The deny creates a conflict with the open policy of 3. This conflict is resolved by the fact that policy was created by the user *Lundy* and has a precedence of 3, where Policy 10 was created by the *network manager* and has a precedence of 1. This results in the fact that Policy 3 is applied to only one of the *paths* generated from the expanded target element, namely *NPS*→*NASA*→*IETF*→*DARPA*→*CERT*. The overlapping path of Policy 3 which is in conflict with Policy 3, is not allowed to carry all traffic.

Conflict 8: The paths specified in Policy 2 and Policy 4 have an overlapping link of *NASA*→*IETF* which produces a conflict. The common link is permitted to carry *all traffic* classes of data with no restrictions from Policy4, at the same time not permitted to carry *NSF* traffic. This produces a conflict that can not be resolved with the *conditions* element of the policy rules. The creator of Policy 2 is the user *Stone* and has a precedence of 4, Policy 4 is created by the *network manager* and has the highest precedence, 1. This resolution of the conflict will nullify the Policy 2 in favor of Policy 4.

Conflict 9: As explained with conflict 6, Policy 8 applies to 8 different *paths* through the network. This policy conflicts with that of Policy 5, which shares a common link of *NPS*→*DARPA*. Policy 8 permits *all traffic* from two *address* groupings. Policy 5 denies *university traffic*, no matter what the *address*, after the hour of 1100. This produces a conflict by permitting and denying *university traffic* after the hour of 1100, for the *address* groupings 153.20.8.* and 131.40.*.*.

The conflict is resolved by disallowing Policy 8 to be enforced on 4 of the paths that it applies to. Policy 5 was given precedence over Policy 8 as a result of the different users creating the policies.

Conflict 10: One of the paths applying Policy 3 is in conflict with the link *NPS→DARPA* of Policy 5. The overlapping *paths* provide a conflict because Policy 3 permits *all traffic* unconditionally, and Policy 5 restricts traffic by not allowing *university traffic* after 1300. The creator of Policy 3 has a lower precedence than the *network manager* who created Policy 5. The conflict resolution disallows Policy 3 to be enforced on the path *NPS→DARPA→CERT*, but will be allowed on the path *NPS→NASA→IETF→DARPA→CERT*, which was also specified in Policy 3.

Conflict 11: Policy 8 produces 8 paths, 4 of which overlap and produce a policy conflict with the link *NPS→DARPA*, specified in Policy 6. Policy 8 permits *all traffic* classes of data as long as they are from a selected list of host addresses. Policy 6 in turn denies *university traffic* for all host *addresses* after the hour of 1300. This produces a policy conflict that results in the 4 overlapping *paths* specified in Policy 8 from being enforced. The remaining 4 paths of Policy 8 produce no conflict with Policy 6 and therefore are allowed to co-exist in the network. The resolution of this policy conflict was a result that Policy 3 was created by a user with a lower precedence. The first 4 paths listed below will not enforce Policy 3. The remaining 4 paths do not produce a conflict and will be allowed.

UN→NATO→NSF→**DARPA**→NPS
UN→NATO→SPAWAR→**DARPA**→NPS
UN→NATO→NSF→SPAWAR→**DARPA**→NPS
UN→NATO→SPAWAR→NSF→**DARPA**→NPS

UN→NATO→NSF→DARPA→IETF→NASA→NPS
UN→NATO→SPAWAR→DARPA→IETF→NASA→NPS
UN→NATO→NSF→SPAWAR→DARPA→IETF→NASA→NPS
UN→NATO→SPAWAR→NSF→DARPA→IETF→NASA→NPS

Conflict 12: Policy 3 and Policy 6 produce a conflict as a result of a overlapping link which both permits and denies the same traffic. This overlapping link is *NPS→DARPA*, and is specified in Policy 6. One of the two *paths* specified in Policy 3, *namely NPS→DARPA→CERT* utilizes the same *NPS→DARPA* link. Policy 3 allows unconditional passage of *all traffic*, where Policy 6 denies *university traffic* after the hour of 1300. This produces a conflict that is resolved by not permitting Policy 3 to be enforced on the overlapping *path segment* listed above, but will allow Policy 3 to be enforced on the path *NPS→NASA→IETF→DARPA→CERT*. The automatic resolution is a result of the fact that a lower precedence user created Policy 3.

Conflict 13: The last policy conflict detected was between Policy 7 and Policy 8. As explained earlier, particularly with conflict 6, the expansion of the path specified in Policy 8 covers a majority of the links in the network by producing 8 unique paths through the network. Policy 7 is specified on a single node (*NSF*) in the network, and as can be seen in paths listed below overlaps with six paths covered by Policy 8.

UN→NATO→NSF→DARPA→NPS
UN→NATO→NSF→SPAWAR→DARPA→NPS
UN→NATO→SPAWAR→NSF→DARPA→NPS
UN→NATO→NSF→DARPA→IETF→NASA→NPS
UN→NATO→NSF→SPAWAR→DARPA→IETF→NASA→NPS
UN→NATO→SPAWAR→NSF→DARPA→IETF→NASA→NPS

UN→NATO→SPAWAR→DARPA→NPS
UN→NATO→SPAWAR→DARPA→IETF→NASA→NPS

The two policies conflict because of the fact that Policy 8 permits *all traffic* when from a particular *address* grouping of hosts. The *addresses* permitted can begin with either 153.20.8, or 131.40. Policy 7 in turn denies *all traffic* from a host that has an *address* beginning with 131. This *address* selection specified in the *conditional* element of the policy rule causes an overlap whenever the *address* belongs to 131, since any *address* starting with 131.40 also starts with 131. This is direct conflict and forces the 6 *paths* that include the node *NSF* to not enforce Policy 8. This results with only 2 of the 8 *paths* specified in Policy 8 to actually support the policy of permitting traffic. The creator of Policy 8 has a precedence lower than that of the *network manager*, who created Policy 7, resulting in the automatic resolution of this conflict.

E. CHAPTER SUMMARY

In this chapter, a case study is presented as a proof of concept that network policies can be represented in PPL, and with the aid of formal logic, conflicts between network policies can be identified. Eleven policies are applied to the ten node network with varying degrees of overlap between those policies. Twenty-nine conflicts were identified between the policies and each conflict is examined in detail.

VI. CONCLUSIONS

A. CASE STUDY CONCLUSIONS

The case study demonstrated that an unambiguous language which supports both path and non-path network policies could indeed represent the structure of a network, as well as the policies to control the operations of that network. It was demonstrated that this can be used to specify static as well as dynamic policies, both of which are needed to support the changing policies of today's networks. Network policies frequently change in response to new requirements, and therefore require continuously re-testing. This need to specify new policies and test their consistency is the reason PPL and the policy tester were developed.

In addition, when multiple network policies were composed together, the result of the compilation was output indicating policies that if implemented would produce conflicting results. The ability to provide a network administrator with information about conflicting policies as early as possible is a major goal of the research in this thesis, and influenced the design of PPL.

Policies were applied to simulated networks varying from 10 to 40 nodes providing feedback to the feasibility of using PPL to support the representation and conflict detection of network policies. The use of a wildcard character in the representation of policy paths had a tremendous impact on the performance of the simulations. When wildcards are present, all possible paths in the network are generated to support the expansion of wildcard paths. The time to process policies for a 10 node network averaged 15 seconds, and for a 40 node network, 27 minutes and 24 seconds. To make the conflict detection process more efficient when wildcard characters are not used, a compilation flag was implemented, "-nowild", that would only include paths explicitly represented by the user. This dropped the compilation time for a 40 node network from over 27 minutes to 7 seconds.

B. CONTRIBUTIONS

While languages have been developed to support specific aspects of network policy, such as Integrated Service, Differentiated Service, and Policy Routing, PPL includes much if not all of the functionality of these specialized languages. PPL represents network policies in an abstract and unambiguous manor. The abstract nature of PPL transcends the issues associated with

platform-dependent network technology, allowing us to specify functionality rather than implementation detail. The benefit of a unambiguous language is that there is no question as to the meaning of the policy's goal.

PPL also provides for the support of static as well as dynamic policies. This enables traffic belonging to certain classes to flow through a network segment, or to provide higher priority to certain types of traffic, such as voice or video. The dynamic aspect of PPL will support the fluctuation of today's network traffic measured by such functions as data loss, jitter, and delay. PPL provides for specifying network policies based on the feedback of such measurements, and controls the tuning of a network as it progresses from state to state. PPL also provides for the user to specify other measurement functions as needed, to enhance the PPL language's power.

It is critical that a network perform in a predictable manor. The ability to detect specific network policies that are in conflict with each other greatly aids in this goal. Although conflict testing is not part of a formal language specification, PPL was designed with conflict detection in mind. A supporting compiler was developed which provides policy testing by utilizing a program written in the Prolog programming language. After parsing the network policies for correctness, the compiler translates the network policies into formal logic, and utilizing the Prolog interpreter, detects conflicts between policies. This feedback can then be utilized by a network administrator to modify the conflicting policies before they are implemented in the network.

The PPL compiler also utilizes the creator of the policy to try and resolve any policy conflicts that have been detected. The allows conflicting policies to co-exist in the specification, but only the higher priority policy will be implemented in the network. The policies that are automatically resolved will also be presented to the user at the end of the compilation process. This feedback provides a sanity check by a human before the policies are implemented throughout the network.

C. LIMITATIONS

Although the PPL language provides the ability to support user-defined types, classes, and messages, there are limitations to predefined attributes. In particular the *hostIP* attribute used in the *conditions* element of a policy rule. *HostIP* is intended to represent an IP address used in the Internet today. Although the dot quad notation used for the *hostIP* attribute is of similar

representation to real IP addressing scheme, it is not complete. The wild carding is limited, and there is no support for subnetting or netmasks. The larger issue is that only the IP version for address schema is supported. There is a need for user defined addresses schemes as well.

The support of dynamic policies is realized with PPL by allowing for user-defined measurement functions for each link, as well as the message that will provide the information. Currently a *message* is defined by providing a name that it can be identified with. There is no sense of contents of the *message*, and no way to represent the possible formats.

Another limitation is with the conflict detection process. In its current state, only the static aspects of network policies are used. This shows that the concept of network policy conflict detection is possible, but is limited until the support of the dynamic aspects of the PPL language are enhanced, specifically *messages*. To support conflict detection with *messages*, a define statement will need to be implemented in the PPL language that allows for the declaration of the return value type and units associated with the message. For example, `delay() <= 20%` or `available_bandwidth() = 24Mbps`. Once the value type and unit of measurements are known, a decision table can be implemented and associated with the decision process.

D. FUTURE DIRECTIONS

The PPL language and compiler provides for the representation, verification and conflict detection of network policies. To actually implement the network policies, the various network devices must be instructed on what aspects of the policies they are to support. Future work could provide tasking files which contain the setup information necessary to program the device. The idea of PPL was to develop an abstract language to support multiple vendors. This trend could be continued with the setup files. Only high-level details of the device setup could be provided. These high level setup instructions might be created using the category of device as a guide. For example, the setup files could be created based on the functions of the device, router, filter, firewall, etc. The setup files for routers would contains the information necessary to create routing tables, but not the manufacturers exact commands.

The approach used to design PPL was that all the network policies are known ahead of time, and that a network administrator would be able to compile and test for consistency before the policies were implemented. A slight modification could be made to accept new network policies and determine if they should be implemented dynamically. For example, in today's

network, resource reservation messages such as RSVP, are used to request resources dynamical, as needed. To create a policy engine that is always available to request the implementation of new policy rules is another possible consideration. The policy engine would provide feedback to the requestor if the policy creates a conflicting situation in the network, is denied because of limited resources, and could generate the setup files needed to dynamically reprogram network devices to support the request.

The next logical step would be to specify and enforce network policies represented in PPL using a testbed network. A network such as SAAM[38], being developed at the Naval Postgraduate School, would be ideal. The SAAM effort is being developed to support QoS in the Next Generation Internet (NGI). A path-based approach is being explored to support the end-to-end QoS needs of today's networks, and would provide an ideal match for PPL. A policy server/engine could be designed based on the research of this dissertation to accept path-based requests, and provide feedback based on the policies applying to the network.

A more user friendly means of creating network policies may also be developed. Currently PPL policy rules are typed into a data file and compiled with a command line interface. The PPL grammar is provided in appendix A, and would provide an excellent starting point for a Graphical User Interface (GUI) tool kit. The tool kit might have a natural field-by-field interface that would provide a syntax-free policy rule. Using input from previous fields of a policy rule, only valid choices would be presented to the user for completion of the policy rule.

A user interface would also provide a security mechanism that would disallow direct manipulation of the PPL configuration file. If the interface was password protected, policies could only be associated with the user currently logged on. This would prevent an ordinary user from assigning the "network managers" identifier to a policy in hopes of guaranteeing its implementation.

Providing a means for the user to order and view existing policies would also be useful. The ordering may provide a view based on the policy creator, or perhaps all the policies that effect a particular network segment. When a policy is applied to several network paths, there is always a chance that a sub-set of paths conflict with other policies, while the remaining paths are conflict free. Providing a means to visualize this information would also be useful feature.

Network managers and planners utilize network simulation tools to predict network performance. If a policy module were developed that could be incorporated in to such tools,

network policies could be applied to network simulations in an attempt to verify network security and performance.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PATH-BASED POLICY LANGUAGE (PPL)

1. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

ALLOW
BITES
BPS
BW
BYTES
CLASS
DEFINE
DENY
GBPS
HOPCOUNT
HOSTIP
KBPS
LINK
LPARAM
MBPS
MBYTES
MESSAGE
MSEC
NODE
NPARAM
PACKETS
PATH
PERMIT
POLICY_MAKER
PPARAM
PRIORITY
SEC
TIME
TYPE
USEC
USERID

These keywords are represented in upper case, but are not case sensitive.

2. Define Statements

To make the language as dynamic as possible, the ability has been provided to the user to expand the functionality of the language with the use of define statements. Define statements such as *node* and *link* are also used as basic building blocks of language. As graph theory uses vertices and edges to define a graph, PPL used *node* and *link* to define the network that policies are to be applied to.

```
define node
define link
define path
define policy_maker
define class
define type
define node_param
define link_param
define path_param
```

3. Policy ID

Policy identifiers are legal variables that identify policies. They are required to be unique and are used to identify policies that are in conflict.

4. User ID

User identifiers are used to identify the creators of policies. They must be unique and be legal variables. These identifiers can be used to prioritize policies that are in conflict. Enabling prioritization is one way to resolve conflicts in network policy.

5. Paths

Paths are the component of policies in PPL that conditions, restrictions, and actions are based on. Paths are composed of at least two nodes in a network that are connected by one or more links. A path can be represented by a list of nodes, a link, a list of links, or a composition of nodes and links. For example, below are valid representations of paths. Assume that node1, node2, and node3 are valid nodes in the network. Also assume that link1 is a link between node2 and node3.

```
{node1, node2}
```

{link1}

{node1,link1}

Nodes may also include wildcard characters. This allows for the representation of multiple paths with a single path statement. For example, the path {*, *node1*, *} represents all possible paths in the network that traverse the node labeled *node1*.

6. Target

The target field of a policy is used to identify a class of traffic. This allows a flow of packets to be identified by a label and is critical for the implementation of differentiated services. Below are examples of how the target component can be used:

{traffic_class == {video, voice}}

{traffic_type == {research}}

7. Conditions

The conditions field is to represent how global conditions can be used to effect a policy. The conditions can either be language defined or user defined. For example *time* is a language defined condition that can be used to restrict a policy to hours between 8:00 am and 4:00pm:

{time >= 0800, time <= 1600}

Conditions can also be user defined, for example let us suppose the user defines a class called *day*, which consists of the names of the days of the week. Then a condition that restricts a policy to weekdays can be represented by:

{day >= Monday, day <= Friday}

8. Action Items

The action item field is used to represent actions that should be taken if the *target* and *condition* fields are satisfied. Action items might be used to set or reset a priority on a traffic class, declaring compromises, and explicit deny actions.

9. Legal Variables

In PPL legal variables are made up of letters, digits and the underscore character. All variables must start with a letter.

10. Dot Quad notation

Dot quad notation is used to represent IP addresses or IP networks that a policy effects. Examples of dot quad notation that can be used are below:

131.*.*
131.1.*.*
131.1.20.*
131.1.20.24

11. Reserved Symbols

The following are reserved symbols that can be used in class membership and conditional statements.

">="

"<="

"!="

".="

"=="

12. Comments

Comments are used to make the program more understandable by allowing the writer to explain briefly what the program does. Comments are any characters between character sequences `/*` and `*/`, and are ignored by the compiler. For example:

```
/* This is a comment */  
  
/*  
 * This is also a comment  
*/
```

13. Formal Grammar

<code><system_policy></code>	<code>:= <define_list> <policy_list> <define_list> <policy_list></code>
<code><define_list></code>	<code>:= <define_type> <define_list> <policy_list></code>
<code><define_type></code>	<code>:= DEFINE <class_define_list> DEFINE <type_define_list> DEFINE <path_define_list> DEFINE <node_define_list> </code>

```

        DEFINE <link_define_list> |
        DEFINE <user_define_list>
        DEFINE <param_define_list>

<class_define_list>      := <class_define_section>
                        | <class_define_list> <class_define_section>
<class_define_section> := CLASS <legal_var> { <class_element_list> } ;
<class_element_list>   := <class_element> | <class_element_list> , <class_element>
<class_element>        := <legal_var>
<function_define_list> := <function_define_section>
                        | <function_define_list> <function_define_section>
<type_define_list>     := <type_define_section>
                        | <type_define_list> <class_define_section>
<type_define_section> := TYPE <legal_var> { <type_element_list> } ;
<type_element_list>   := <type_element>
                        | <type_element_list> , <type_element>
<type_element>        := <legal_var>
<path_define_list>     := <path_define_section>
                        | <path_define_list> <path_define_section>
<path_define_section> := PATH <legal_var> { <define_path_list> } ;
<define_path_list>     := <define_path>
                        | <define_path_list> , <define_path>
<define_path>          := < * > | * | < <define_path_element_list> >
<define_path_element_list> := <define_path_element> , <define_path_element>
                        | <define_path_element_list> , <define_path_element>
<define_path_element> := <define_path_element> , <define_path_element>
                        | <define_path_element_list> , <define_path_element>
<define_path_element> := <legal_var> | *
<node_define_list>     := NODE <node_list> ;
<node_list>            := <node> | <node_list> , <node>
<node>                 := <legal_var>
<link_define_list>     := LINK <link_list> ;

```


<link_list>	:= <link> <link_list> , <link>
<link>	:= <legal_var> < <legal_var> , <legal_var> >
<user_define_list>	:= POLICY_MAKER <user_list> ;
<user_list>	:= <define_user> <user_list> , <define_user>
<define_user>	:= <legal_var> (<integer>)
<param_define_list>	:= NODE_PARAM <legal_var> { <param_list> } ; LIST_PARAM <legal_var> { <param_list> } ; PATH_PARAM <legal_var> { <param_list> } ;
<param_element>	:= BW := <integer> <bw_unit> <legal_var> ()
<param_list>	:= <param_element> <param_list> , <param_element>
<policy_list>	:= <policy> <policy_list> <policy>
<policy>	:= <policyID> <userID> { <path_list> } { <target_list> } { <condition_list> } { <action_list> } ;
<policyID>	:= <legal_var>
<userID>	:= <legal_var>
<path_list>	:= <defined_path> <path_list> , <defined_path>
<defined_path>	:= < legal_var >
<target_list>	:= <target> <target_list> , <target> *
<target>	:= <legal_var> <target_symbol> { <target_element_list> }
<target_element_list>	:= <target_element> <target_element_list> , <target_element>
<target_element>	:= <legal_var>
<target_symbol>	:= == !=
<condition_list>	:= * <condition> <condition_list> , <condition>
<condition>	:= PRIORITY <= <integer> PRIORITY >= <integer> HOPCOUNT >= <integer>

| **HOPCOUNT** <= <integer>
 | **TIME** <= <time>
 | **TIME** >= <time>
 | **HOSTIP** = <quad_dot>
 | **HOSTIP** != <quad_dot>
 | <bandwidth_symbol> >= <number_bw_units>
 | <bandwidth_symbol> <= <number_bw_units>
 | <legal_var> <conditional_defined_operations>
 <conditional_defined_operations> := <conditional_operations>
 | () <conditional_symbol> <number_units>
 <conditional_symbol> := **GTEQ** | **LSEQ** | **NTEQ** | **EQUAL** | < | >
 <bandwidth_symbol> := **BW**
 <conditional_operations> := <conditional_symbol> <legal_var>
 <action_list> := **DENY**
 | <action> | <action_list> , <action>
 <action> := <action_reserved_word>
 | <action_reserved_word> := <number_units>
 <action_reserved_word> := **PRIORITY** | **PERMIT** | **HOPCOUNT**
 <bw_unit> := **GBPS** | **MBPS** | **KBPS** | **BPS**
 <time> := <integer> | <integer> : <integer>
 <number_units> := <number> | <number> <unit>
 <number> := <integer> | <float>
 <number_bw_units> := <output_number> <bw_unit>
 <output_number> := <integer> | <float>
 <quad_dot> := <quad_dot_1> |
 <quad_dot_2> |
 <quad_dot_3> |
 <quad_dot_4>
 <unit> := **MBYTES** | **BYTES** | **BITES** | **SEC** |
MSEC | **USEC** | **PACKETS**

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. POLICY RULE INPUT FILE OF CASE STUDY

```
/*
 * Define the nodes of the network
 */
define node NPS, DARPA, SPAWAR, NSF, IETF, Internet_link1, Internet_link2;

/*
 * Define the links of the network
 */
define link NPS_DARPA <NPS, DARPA>, NPS_NASA <NPS, NASA>;
define link DARPA_IETF <DARPA, IETF>, DARPA_NSF <DARPA, NSF>;
define link DARPA_SPAWAR <DARPA, SPAWAR>, DARPA_CERT <DARPA, CERT>;
define link NASA_IETF <NASA, IETF>;
define link IETF_FDA <IETF, FDA>;
define link SPAWAR_NSF <SPAWAR, NSF>, SPAWAR_NATO <SPAWAR, NATO>;
define link NSF_NATO <NSF, NATO>;
define link UN_NATO <UN, NATO>;

/*
 * Define the paths used in the network policies
 */
define path NPS_CERT {<NPS, *, CERT>;};
define path NPS_NSF {<NPS, DARPA, NSF>;};
define path NASA_SPAWAR {<NASA, IETF, DARPA, SPAWAR>;};
define path UN_NPS {<UN, *, NPS>;};
define path ALL {*};

/*
 * Define the users who can create policies
 */
define policy_maker Net_Manager(1), Xie(3), Lundy(3), Stone(5);

/*
 * User defined classes of traffic which can be used
 * in the target element of policy rules
 */
define class traffic_type {research, university};
define class traffic_class {data, video, voice};
define class traffic_security {Private, Public};
define class traffic_priority {High, Med, Low};
define class user {faculty, student, staff, accounting};
define class node_traffic {NPS, DARPA, SPAWAR, NSF, IETF, Internet_link1, Internet_link2};
```

```

/*
 * User defined type that can be used in the conditional element
 * of a policy rule.
 */
define day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

/*
 * Paramaters associated with links and paths
 */
define link_param NPS_DARPA {BW := 100MBPS, delay(), loss_rate()};
define path_param NPS_CERT {BW := 100 MBPS, jitter()};
define path_param NPS_NSF {BW := 100 MBPS, jitter()};

/*
 * Format of Policy Term:
 * PolicyID UserID {paths} {target} {conditions} {action_items};
 */

/*
 * Permit research, NPS and NSF traffic Monday - Friday
 * between the hours of 0800 and 1200
 */
Policy1 Net_Manager {NPS_NSF}
    {traffic_type == {research}, node_traffic == {NPS, NSF}}
    {day != Saturday, day != Sunday, time >= 0800, time <= 1200}
    {permit};

/*
 * Deny all traffic from NSF on the link between NASA and SPAWAR
 */
Policy2 Stone {NASA_SPAWAR} {node_traffic == {NSF}} {*} {deny};

/*
 * Permit all traffic unconditionally on the two paths
 * specified with the NPS_CERT path definition
 */
Policy3 Lundy {NPS_CERT} {*} {*} {permit};

/*
 * Permit all traffic unconditionally on the NASA->IETF link
 */
Policy4 Net_Manager {NASA_IETF} {*} {*} {permit};

```

```

/*
 * Deny all university traffic after 1100 on the
 * NPS->DARPA link
 */
Policy5 Net_Manager {NPS_DARPA}
    {traffic_type == {university}}
    {time >= 1100}
    {deny};

/*
 * Deny university traffic after 1300 on the
 * NPS->DARPA link
 */
Policy6 Net_Manager {NPS_DARPA}
    {traffic_type == {university}}
    {time >= 1300}
    {deny};

/*
 * Deny all traffic from hosts with and address beginning
 * with 131.
 */
Policy7 Net_Manager {NSF} {*} {hostIP == 131.*.*} {deny};

/*
 * Permit all traffic from hosts whos addresses begin with
 * either 153.20.8 or 131.40.
 */
Policy8 Xie {UN_NPS} {*} {hostIP == 153.20.8.*, hostIP == 131.40.*.*} {permit};

/*
 * Permit and assign a priority of 3 to video traffic before the hour
 * of 0800, on the NPS->DARPA link
 */
Policy9 Net_Manager {NPS_DARPA} {traffic_class == {video}}
    {time <= 0759} {priority := 3};

/*
 * Deny video traffic on the NPS->DARPA link between the hours of 0800 and 1600
 */
Policy10 Net_Manager {NPS_DARPA} {traffic_class == {video}}
    {time >= 0800, time <= 1600} {deny};

```

```
/*  
 * Permit and assign a priority of 5 to video traffic after the hour  
 * of 1600 on the NPS->DARPA link  
 */  
Policy11 Net_Manager {NPS_DARPA} {traffic_class == {video}}  
        {time >=1601} {priority := 5};
```

APPENDIX C. POLICY RULE OUTPUT FILE FOR CASE STUDY

Print Unresolved conflicts

Conflict Policy10 \Longleftrightarrow Policy1

Policy10 Path = [NPS,DARPA]

Policy10 Targets: deny traffic_class=[video]

Policy1 Path = [NPS,DARPA,NSF]

Policy1 Targets: permit traffic_type=[research], permit node_traffic=[NPS], permit node_traffic=[NSF]

Target Conflicts: node_traffic = [NPS], node_traffic = [NSF], traffic_type = [research]

Conflict Policy5 \Longleftrightarrow Policy11

Policy5 Path = [DARPA,NPS]

Policy5 Targets: deny traffic_type=[university]

Policy11 Path = [DARPA,NPS]

Policy11 Targets: permit traffic_class=[video]

Target Conflicts: traffic_class = [video]

Conflict Policy5 \Longleftrightarrow Policy1

Policy5 Path = [NPS,DARPA]

Policy5 Targets: deny traffic_type=[university]

Policy1 Path = [NPS,DARPA,NSF]

Policy1 Targets: permit traffic_type=[research], permit node_traffic=[NPS], permit node_traffic=[NSF]

Target Conflicts: node_traffic = [NPS], node_traffic = [NSF], traffic_type = [research]

Conflict Policy5 \Longleftrightarrow Policy11

Policy5 Path = [NPS,DARPA]

Policy5 Targets: deny traffic_type=[university]

Policy11 Path = [NPS,DARPA]

Policy11 Targets: permit traffic_class=[video]

Target Conflicts: traffic_class = [video]

Conflict Policy6 \Longleftrightarrow Policy11

Policy6 Path = [DARPA,NPS]

Policy6 Targets: deny traffic_type=[university]

Policy11 Path = [DARPA,NPS]

Policy11 Targets: permit traffic_class=[video]

Target Conflicts: traffic_class = [video]

Conflict Policy6 \Longleftrightarrow Policy11

Policy6 Path = [NPS,DARPA]
Policy6 Targets: deny traffic_type=[university]
Policy11 Path = [NPS,DARPA]
Policy11 Targets: permit traffic_class=[video]
Target Conflicts: traffic_class = [video]

Conflict Policy7 <==> Policy1
Policy7 Path = [NSF]
Policy7 Targets: []
Policy1 Path = [NPS,DARPA,NSF]
Policy1 Targets: permit traffic_type=[research], permit node_traffic=[NPS], permit
node_traffic=[NSF]
Target Conflicts: node_traffic = [NPS], node_traffic = [NSF], traffic_type = [research]

Print Resolved conflicts

Conflict Policy10 <==> Policy8
Policy10 Path = [DARPA,NPS]
Policy10 Targets: deny traffic_class=[video]
Policy8 Path = [UN,NATO,NSF,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy10(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy10 <==> Policy8
Policy10 Path = [DARPA,NPS]
Policy10 Targets: deny traffic_class=[video]
Policy8 Path = [UN,NATO,NSF,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy10(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy10 <==> Policy8
Policy10 Path = [DARPA,NPS]
Policy10 Targets: deny traffic_class=[video]
Policy8 Path = [UN,NATO,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy10(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy10 <==> Policy8
Policy10 Path = [DARPA,NPS]
Policy10 Targets: deny traffic_class=[video]
Policy8 Path = [UN,NATO,SPAWAR,NSF,DARPA,NPS]
Policy8 Targets: []

Target Conflicts: Policy8 = [permit_all]
Resolved: Policy10(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy10 <==> Policy3
Policy10 Path = [NPS,DARPA]
Policy10 Targets: deny traffic_class=[video]
Policy3 Path = [NPS,DARPA,CERT]
Policy3 Targets: []
Target Conflicts: Policy3 = [permit_all]
Resolved: Policy10(Priority = 1) overrides=> Policy3(Priority = 3)

Conflict Policy4 <==> Policy2
Policy4 Path = [NASA,IETF]
Policy4 Targets: []
Policy2 Path = [NASA,IETF,DARPA,SPAWAR]
Policy2 Targets: deny node_traffic=[NSF]
Target Conflicts: Policy4 = [permit_all]
Resolved: Policy4(Priority = 1) overrides=> Policy2(Priority = 4)

Conflict Policy5 <==> Policy8
Policy5 Path = [DARPA,NPS]
Policy5 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,NSF,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy5(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy5 <==> Policy8
Policy5 Path = [DARPA,NPS]
Policy5 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,NSF,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy5(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy5 <==> Policy8
Policy5 Path = [DARPA,NPS]
Policy5 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy5(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy5 <==> Policy8
Policy5 Path = [DARPA,NPS]
Policy5 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,SPAWAR,NSF,DARPA,NPS]

Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy5(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy5 <==> Policy3
Policy5 Path = [NPS,DARPA]
Policy5 Targets: deny traffic_type=[university]
Policy3 Path = [NPS,DARPA,CERT]
Policy3 Targets: []
Target Conflicts: Policy3 = [permit_all]
Resolved: Policy5(Priority = 1) overrides=> Policy3(Priority = 3)

Conflict Policy6 <==> Policy8
Policy6 Path = [DARPA,NPS]
Policy6 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,NSF,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy6(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy6 <==> Policy8
Policy6 Path = [DARPA,NPS]
Policy6 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,NSF,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy6(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy6 <==> Policy8
Policy6 Path = [DARPA,NPS]
Policy6 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy6(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy6 <==> Policy8
Policy6 Path = [DARPA,NPS]
Policy6 Targets: deny traffic_type=[university]
Policy8 Path = [UN,NATO,SPAWAR,NSF,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy8 = [permit_all]
Resolved: Policy6(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy6 <==> Policy3
Policy6 Path = [NPS,DARPA]

Policy6 Targets: deny traffic_type=[university]
Policy3 Path = [NPS,DARPA,CERT]
Policy3 Targets: []
Target Conflicts: Policy3 = [permit_all]
Resolved: Policy6(Priority = 1) overrides=> Policy3(Priority = 3)

Conflict Policy7 <=> Policy8
Policy7 Path = [NSF]
Policy7 Targets: []
Policy8 Path = [UN,NATO,NSF,DARPA,IETF,NASA,NPS]
Policy8 Targets: []
Target Conflicts: Policy7 = [deny_all]
Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy7 <=> Policy8
Policy7 Path = [NSF]
Policy7 Targets: []
Policy8 Path = [UN,NATO,NSF,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy7 = [deny_all]
Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy7 <=> Policy8
Policy7 Path = [NSF]
Policy7 Targets: []
Policy8 Path = [UN,NATO,NSF,SPAWAR,DARPA,IETF,NASA,NPS]
Policy8 Targets: []
Target Conflicts: Policy7 = [deny_all]
Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy7 <=> Policy8
Policy7 Path = [NSF]
Policy7 Targets: []
Policy8 Path = [UN,NATO,NSF,SPAWAR,DARPA,NPS]
Policy8 Targets: []
Target Conflicts: Policy7 = [deny_all]
Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy7 <=> Policy8
Policy7 Path = [NSF]
Policy7 Targets: []
Policy8 Path = [UN,NATO,SPAWAR,NSF,DARPA,IETF,NASA,NPS]
Policy8 Targets: []
Target Conflicts: Policy7 = [deny_all]
Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy7 \Longleftrightarrow Policy8

Policy7 Path = [NSF]

Policy7 Targets: []

Policy8 Path = [UN,NATO,SPAWAR,NSF,DARPA,NPS]

Policy8 Targets: []

Target Conflicts: Policy7 = [deny_all]

Resolved: Policy7(Priority = 1) overrides \Rightarrow Policy8(Priority = 3)

APPENDIX D. PPL PARSER SOURCE CODE

```
/* INCLUDE SECTION */

%{
#include "symbol.h"
#include <stdio.h>
#include <string.h>

/* Function prototype to print out the
 * compiler user options
 */
void print_usage();

/*
 * Text string passed in by flex
 */
extern char *yytext;

/*
 * Pointer to output file created to contain
 * formal logic statements
 */
FILE *logic_out;

/*
 * Variables used record the compiler options
 * entered by the user
 */
int no_wild = 0;
int no_implicit_deny = 0;
extern int verbose;

/*
 * Temporary Pointers to entries in Symbol Table
 */
struct symtab *class_id = 0;
struct symtab *function_id = 0;
struct symtab *type_id = 0;
struct symtab *path_id = 0;
struct symtab *param_id = 0;
struct symtab *policy_id = 0;
struct symtab *link_name = 0;
struct symtab *link_node1 = 0;
```

```

struct symtab *link_node2 = 0;

/* Return value from symbol lookup function */
int class_found;

/* Flag to indicate the first time in the loop */
int first_time;

/*
 * Flag to indicate if defined type is to be
 * represented in formal logic, and output to file.
 */
int output_flag = 0;

/* Number of policies processed */
int policy_count = 0;

/*
 * Flag to indicate messages supported on network links
 * are to output to the file.
 */
int message_flag = 0;

/* Count to uniquely identified defined types internally */
int type_count = 0;

/*
 * Variables to hold clock times converted from strings to
 * integers
 */
int clock_time, clock_time1, clock_time2;

%}

/* DEFINITION SECTION */

/*
 * Union to hold possible symbol types
 */
%union {
    struct symtab *symp;
    char *string;
}

/* Define the following Variables as type string */
%token <string> LEGALVAR
%token <string> INTEGER

```

```

%token <string> FLOAT
%token <string> QUAD_DOT_1
%token <string> QUAD_DOT_2
%token <string> QUAD_DOT_3
%token <string> QUAD_DOT_4

/*
 * Reserved Symbols
 */
%token GTEQ LSEQ NTEQ ASSIGN EQUAL CR WILDCARD

/*
 * Reserved Words
 */
%token DENY PRIORITY ALLOW PERMIT HOPCOUNT HOSTIP USERID
%token TIME DAY DEFINE CLASS MESSAGE TYPE PATH NODE LINK
POLICY_MAKER
%token NPARAM LPARAM PPARAM
%token BW

/*
 * Measurement Units
 */
%token MBYTES BYTES GBPS MBPS Kbps BITES BPS SEC MSEC USEC PACKETS

/* RULES SECTION */
%%

system_policy: define_list
               | policy_list
               | define_list policy_list
               ;

/*
 * Rules to allow user defined classes, messages, types, paths,
 * nodes, links, and users
 */

define_list:  define_type
              | define_list define_type
              ;

define_type:  DEFINE class_define_list
              | DEFINE type_define_list
              | DEFINE path_define_list
              | DEFINE node_define_list
              | DEFINE link_define_list

```



```

        | DEFINE user_define_list
        | DEFINE param_define_list
        ;

/*
 * Rules for defining parameters associated with nodes, links, and paths
 */

param_define_list:      NPARAM LEGALVAR
                        {if (sym_look ($2,DEFINED_NODE) != FOUND_ENTRY)
                          {yyerror ("Unknown NODE ");}
                        else {param_id = sym_tab_ref ($2, DEFINED_NODE);
                          message_flag = 0;};}
                        '{' param_list '}' ';'
        | LPARAM LEGALVAR
                        {if (sym_look ($2,DEFINED_LINK) != FOUND_ENTRY)
                          {yyerror ("Unknown LINK ");}
                        else {param_id = sym_tab_ref ($2, DEFINED_LINK);
                          message_flag = 1;};}
                        '{' param_list '}' ';'
        | PPARAM LEGALVAR
                        {if (sym_look ($2,DEFINED_PATH) != FOUND_ENTRY)
                          {yyerror ("Unknown PATH ");}
                        else {param_id = sym_tab_ref ($2, DEFINED_PATH);
                          message_flag = 0;};}
                        '{' param_list '}' ';'
        ;

param_list:             param_element
        | param_list ',' param_element
        ;

param_element:          BW ASSIGN INTEGER
                        {fprintf(logic_out,"path_param(\'%s\',\'BW\',
                          \'=\'\'%s',(param_id)->name,$3);}
                        bw_unit
                        {fprintf(logic_out,")\n");}
        | LEGALVAR
                        {if (sym_look ($1,DEFINED_MESSAGE) != FOUND_ENTRY){
                          sym_add($1,DEFINED_MESSAGE);}
                        if (sym_tab_entry_lookup(param_id, $1) == NO_ENTRY){
                          sym_tab_entry_add (param_id, $1);}
                        if (message_flag)
                          fprintf(logic_out,"path_message(\'%s\',\'%s\',\n",
                          (param_id)->name,$1);}
                        '(' ')'
        ;

```

```

/*
 * Rules for defining users who can make policies
 */

```

```

user_define_list:    POLICY_MAKER user_list ';
                    ;
user_list:           define_user
                    | user_list ',' define_user
                    ;
define_user:         LEGALVAR
                    {if (sym_look ($1,DEFINED_USER) == FOUND_ENTRY)
                      {yyerror ("Re-definition of USER ")};
                     else {class_id = sym_add($1, DEFINED_USER);};}
                    '(' INTEGER
                    {sym_tab_entry_add(class_id, $4);}
                    ')'
                    ;

```

```

/*
 * Rules for defining links in the network
 */

```

```

link_define_list:    LINK link_list ';
                    ;
link_list:           link
                    | link_list ',' link
                    ;
link:               LEGALVAR
                    {if (sym_look ($1,DEFINED_LINK) == FOUND_ENTRY)
                      {yyerror ("Re-definition of Link ")};
                     else {link_name = sym_add($1, DEFINED_LINK);};}
                    '<' LEGALVAR
                    {if ((sym_look ($4,DEFINED_NODE)) != FOUND_ENTRY)
                      {yyerror ("Unknown Node in Link ")};
                      link_node1 = sym_tab_ref ($4, DEFINED_NODE);}
                    ',' LEGALVAR
                    {if ((sym_look ($7,DEFINED_NODE)) != FOUND_ENTRY)
                      {yyerror ("Unknown Node in Link ")};
                      link_node2 = sym_tab_ref ($7, DEFINED_NODE);}
                    '>'
                    {fprintf(logic_out,"link('%s','%s','%s').\n",
                      (link_name)->name,(link_node1)->name,(link_node2)->name);
                     fprintf(logic_out,"link('%s','%s','%s').\n",
                      (link_name)->name,(link_node2)->name,(link_node1)->name);
                     }
                    ;

```

```

/*
 * Rules for defining nodes in the network
 */

node_define_list:      NODE node_list';
                        ;
node_list:              node
                        |   node_list ',' node
                        ;

node:                   LEGALVAR
                        {if ((class_found = sym_look ($1,DEFINED_NODE)) == FOUND_ENTRY)
                          {yyerror ("Re-definition of Node ");}
                          else {class_id = sym_add($1, DEFINED_NODE);}}
                        ;

/*
 * Rules for defining paths in the network
 */

path_define_list:       path_define_section
                        |   path_define_list path_define_section
                        ;
path_define_section:    PATH LEGALVAR
                        {if ((sym_look ($2,DEFINED_PATH)) == FOUND_ENTRY)
                          {yyerror ("Re-definition of Path ");}
                          else {class_id = sym_add($2, DEFINED_PATH);
                                fprintf (logic_out,"path_links(\"%s\",[\" , $2);}}
                                {' define_path_list '}' ;' {fprintf (logic_out,")\n");}
                          }
                        ;

define_path_list:       define_path
                        |   define_path_list ' ' {fprintf (logic_out,",");} define_path
                        ;

define_path:            '< WILDCARD >' {fprintf (logic_out,"*");}
                        |   WILDCARD {fprintf (logic_out,"*");}
                        |   '< define_path_element_list >'
                        ;

define_path_element_list:  define_path_element ' ' {fprintf (logic_out,",");}
define_path_element
                        |   define_path_element_list ' ' {fprintf (logic_out,",");}
define_path_element
                        ;
define_path_element:     LEGALVAR
                        {if (sym_look ($1, DEFINED_NODE) != FOUND_ENTRY)

```

```

                                {yyerror ("Invalid Path (Node not defined) ");}
                                else {fprintf (logic_out,"%s",$1);;}
                                WILDCARD {fprintf (logic_out,"*");}
                                ;

/*
 * Rules for defining traffic classes to be used in the network
 */

class_define_list:      class_define_section
                        |      class_define_list class_define_section
                        ;

class_define_section:   CLASS LEGALVAR
                        {if ((sym_look ($2,DEFINED_CLASS)) == FOUND_ENTRY)
                            {yyerror ("Re-definition of Class ");}
                        else {class_id = sym_add($2, DEFINED_CLASS);;}
                        '{' class_element_list '}' ';';

class_element_list:     class_element
                        |      class_element_list ',' class_element
                        ;

class_element:   LEGALVAR {sym_tab_entry_add(class_id, $1);};


/*
 * Rules for defining types to be used in the network
 */

type_define_list:      type_define_section
                        |      type_define_list class_define_section
                        ;

type_define_section:   TYPE LEGALVAR
                        {type_count = 0;
                        if ((sym_look ($2, DEFINED_TYPE)) == FOUND_ENTRY)
                            {yyerror ("Re-definition of Class ");}
                        else {class_id = sym_add($2, DEFINED_TYPE);;}
                        '{' type_element_list '}' ';';

type_element_list:     type_element
                        |      type_element_list ',' type_element
                        ;

type_element:   LEGALVAR
                {sym_tab_entry_type_add(class_id, $1, type_count);type_count++;};

```

```

/*
 * Syntax for Policy Term
 * policyID <userID> {paths} {target} {conditions} {action_items}
 * - policyID      - unique policy identification token
 * - userID        - user ID of policy creator
 * - paths         - network paths the policy effects
 * - target        - target class of network traffic
 * - conditions    - any global conditions (items are AND'ed)
 * - action_items - for setting parameters (e.g. policy priority),
 *                  declaring compromises and explicit deny, etc.
 */

/*
 * Rules for defining policies
 */
policy_list:    policy
               |    policy_list policy
               ;

policy: policyID userID '{' path_list '}' '{'
      {fprintf (logic_out, "policy_targets('%s\\',[" , (policy_id)->name);}
      target_list
      {fprintf (logic_out, "]).\\n");}
      '{' '{' condition_list '}' '{' action_list '}' ' ';
      ;

policyID:      LEGALVAR
              {if (sym_look ($1, POLICYID) == FOUND_ENTRY)
                {yyerror ("Re-definition of Policy ID ");}
               else {policy_id = sym_add($1, POLICYID);};policy_count++;}
              ;

userID: LEGALVAR
       {if (sym_look ($1, DEFINED_USER) != FOUND_ENTRY)
         {yyerror ("UNKNOWN user ");}
        else
          {fprintf (logic_out, "policy_owner('%s\\', '%s\\').\\n", (policy_id)->name, $1);};}
       ;

/*
 * Syntax for Path Component
 */

path_list:      defined_path
               |    path_list ',' defined_path
               ;

defined_path:   LEGALVAR

```

```

        {if ((sym_look ($1,DEFINED_PATH) != FOUND_ENTRY) &&
            (sym_look ($1,DEFINED_LINK) != FOUND_ENTRY) &&
            (sym_look ($1,DEFINED_NODE) != FOUND_ENTRY)){
                yyerror ("Unknown DEFINED element ");}
        else{fprintf(logic_out,"policy_path(\'%s\',\'%s\').\n",(policy_id)->name,$1);}}
    ;

/*
 * Syntax for Target Component
 */

target_list:    target {}
               |    target_list ',' {fprintf(logic_out,",");} target
               |    WILDCARD
               |    {fprintf(logic_out,"*','*',[");}
               |    sym_add((policy_id)->name, TARGET_ALL);}
               ;

target:         LEGALVAR
               {if (sym_look($1, DEFINED_CLASS) == FOUND_ENTRY)
                   {class_id = sym_tab_ref($1, DEFINED_CLASS);
                    fprintf(logic_out,"%s", $1);}
               else yyerror ("Invalid Class");}
               target_symbol '{'
               {fprintf(logic_out,"[");}
               target_element_list
               {fprintf(logic_out,"]");}
               '}'
               ;

target_element_list:  target_element {}
                   |    target_element_list ',' {fprintf(logic_out,",");}
                   |    target_element
                   ;

target_element: LEGALVAR
               {if (sym_tab_entry_look( (class_id)->name,$1,
                                         DEFINED_CLASS) == NO_ENTRY)
                   {yyerror ("Invalid Class Entry");}
               fprintf(logic_out,"%s", $1);}
               ;

target_symbol: EQUAL {fprintf(logic_out,"'='");}
              |    NTEQ {fprintf(logic_out,"'!=');}
              ;

/*
 * Syntax for Conditions Component
 */

condition_list: WILDCARD {fprintf(logic_out,"no_conditions(\'%s\').\n",(policy_id)->name);}

```

```

|         {class_id = function_id = type_id = (struct sytab *) -1;}
|         condition {}
|         condition_list ',' {class_id = function_id = type_id = (struct sytab *) -1;}
|         condition {}
;
condition:  PRIORITY LSEQ INTEGER
|           {fprintf(logic_out,"condition_path('%s\'',priority,<=,%s).\n",
|                 (policy_id)->name,$3);}
|           PRIORITY GTEQ INTEGER
|           {fprintf(logic_out,"condition_path('%s\'',priority,>=,%s).\n",
|                 (policy_id)->name,$3);}
|           HOPCOUNT GTEQ INTEGER
|           {fprintf(logic_out,"condition_path('%s\'',hopcount,>=,%s).\n",
|                 (policy_id)->name,$3);}
|           HOPCOUNT LSEQ INTEGER
|           {fprintf(logic_out,"condition_path('%s\'',hopcount,>=,%s).\n",
|                 (policy_id)->name,$3);}
|           TIME LSEQ {fprintf(logic_out,"condition_path('%s\'',time,<=,",
|                 (policy_id)->name);} time
|           TIME GTEQ {fprintf(logic_out,"condition_path('%s\'',time,>=,",
|                 (policy_id)->name);} time
|           HOSTIP EQUAL
|           {fprintf(logic_out,"condition_path('%s\'','host_id\'','==\'',"
|                 (policy_id)->name);}
quad_dot
|           {fprintf(logic_out,")\n");}
|           HOSTIP NTEQ
|           {fprintf(logic_out,"condition_path('%s\'','host_id\'','!=\'',"
|                 (policy_id)->name);}
quad_dot
|           {fprintf(logic_out,")\n");}
|           bandwidth_symbol GTEQ
|           {fprintf(logic_out,"condition_path('%s\'','BW\'',>=,['',(policy_id)->name);}
|           number_bw_units
|           {fprintf(logic_out,")\n");}
|           bandwidth_symbol LSEQ
|           {fprintf(logic_out,"condition_path('%s\'','BW\'',<=,['',(policy_id)->name);}
|           number_bw_units
|           {fprintf(logic_out,")\n");}

|           USERID EQUAL LEGALVAR
|           {fprintf(logic_out,"condition_path('%s\'','user_id\'','==\'','%s\'').\n",
|                 (policy_id)->name,$3);}
|           USERID NTEQ LEGALVAR
|           {fprintf(logic_out,"condition_path('%s\'','user_id\'','!=\'','%s\'').\n",
|                 (policy_id)->name,$3);}
|           LEGALVAR

```

```

{ if (sym_look ($1, DEFINED_MESSAGE) == FOUND_ENTRY){
    function_id = sym_tab_ref ($1, DEFINED_MESSAGE);
    fprintf(logic_out,"policy_message(\"%s\", \"%s\").\n",
        (policy_id)->name,$1);
};
if (sym_look ($1, DEFINED_TYPE) == FOUND_ENTRY){
    type_id = sym_tab_ref ($1, DEFINED_TYPE);
    output_flag = 1;
};
if (class_id == (struct symtab *) -1 &&
    function_id == (struct symtab *) -1 &&
    type_id == (struct symtab *) -1){
    yyerror ("Unknown DEFINED element ");
};
conditional_defined_operations {}
;

```

time: INTEGER

```

{clock_time = atoi($1);
if (clock_time >= 0 && clock_time < 2400){
    if (clock_time < 10){fprintf(logic_out,"000%d00).\n",clock_time);}
    else if (clock_time < 100){fprintf(logic_out,"00%d00).\n",clock_time);}
    else if (clock_time < 1000){fprintf(logic_out,"0%d00).\n",clock_time);}
    else {fprintf(logic_out,"%d00).\n",clock_time);}
}
else {yyerror ("Invalid Time: range 0000-2359 ");}}
| INTEGER ':' INTEGER
{clock_time1 = atoi($1);
clock_time2 = atoi($3);
if (clock_time1 >= 0 && clock_time1 < 2400 ){
    if (clock_time1 < 10){fprintf(logic_out,"000%d",clock_time1);}
    else if (clock_time1 < 100){fprintf(logic_out,"00%d",clock_time1);}
    else if (clock_time1 < 1000){fprintf(logic_out,"0%d",clock_time1);}
    else {fprintf(logic_out,"%d",clock_time1);}
}
else {yyerror ("Invalid Time: range 0000:00-2359:59 ");}
if (clock_time2 >= 0 && clock_time2 <= 59){
    if (clock_time2 < 10){fprintf(logic_out,"0%d).\n",clock_time2);}
    else {fprintf(logic_out,"%d).\n",clock_time2);}
}
else {yyerror ("Invalid Time X:Y : X range 0000-2359, Y range 00-59");}}
;

```

conditional_defined_operations:conditional_operations
| '(')'


```

        {if (function_id == (struct symtab *) -1) {
            yyerror ("Unknown Message");}}
        conditional_symbol number_units
    ;

conditional_operations: {if (output_flag){
    fprintf(logic_out,"condition_path(\'%s\',\'%s\',",
    (policy_id)->name,(type_id)->name);
    }}
    conditional_symbol LEGALVAR
    {if (class_id == (struct symtab *) -1 &&
        type_id == (struct symtab *) -1){
        yyerror ("Invalid CLASS/TYPE");
    }
    if (class_id != (struct symtab *) -1)
        {if(sym_tab_entry_look((class_id)->name,$3,
            DEFINED_CLASS) == NO_ENTRY){
            yyerror ("Invalid CLASS Member");
        }
    }
    if (type_id != (struct symtab *) -1) {
        if(sym_tab_entry_look((type_id)->name,$3,
            DEFINED_TYPE) == NO_ENTRY){
            yyerror ("Invalid TYPE member");
        }
        else{
            fprintf(logic_out,"\'%s\'.\n",$3);
        }
    }
    }}
;

number_bw_units:    output_number bw_unit;

bw_unit:
    |    GBPS {fprintf(logic_out,"\'GBPS\'");}
    |    MBPS {fprintf(logic_out,"\'MBPS\'");}
    |    KBPS {fprintf(logic_out,"\'KBPS\'");}
    |    BPS {fprintf(logic_out,"\'BPS\'");}
;

number_units:
    |    number
    |    number unit
;

conditional_symbol:    GTEQ {if (output_flag) {fprintf(logic_out,"\'>=\'"); output_flag = 0;}}
    |    LSEQ {if (output_flag) {fprintf(logic_out,"\'<=\'");output_flag = 0;}}
    |    NTEQ {if (output_flag) {fprintf(logic_out,"\'!=\'");output_flag = 0;}}

```

```

|          EQUAL
|          {if (output_flag) {fprintf(logic_out, "'='"); output_flag = 0;}}
|          '>'
|          {if (type_id != (struct sytab *) -1){yyerror ("Invalid Type Operator");}
|          if (output_flag){fprintf(logic_out, "'>"); output_flag = 0;}}
|          '<'
|          {if (type_id != (struct sytab *) -1){yyerror ("Invalid Type Operator");}
|          if (output_flag) {fprintf(logic_out, "'<"); output_flag = 0;}}
;

bandwidth_symbol:    BW {}
;

number:              INTEGER{}
|                   FLOAT{}
;

output_number: INTEGER{fprintf(logic_out, "%s", $1);}
|              FLOAT{fprintf(logic_out, "%s", $1);}
;

quad_dot:            QUAD_DOT_1 {replace_dots($1); fprintf(logic_out, "[%s]", $1);}
|                   QUAD_DOT_2 {replace_dots($1); fprintf(logic_out, "[%s]", $1);}
|                   QUAD_DOT_3 {replace_dots($1); fprintf(logic_out, "[%s]", $1);}
|                   QUAD_DOT_4 {replace_dots($1); fprintf(logic_out, "[%s]", $1);}
;

unit:                '%'
|                   MBYTES
|                   BYTES
|                   BITES
|                   SEC
|                   MSEC
|                   USEC
|                   PACKETS
;

/*
 * Syntax for Action Items Component
 */

action_list:         DENY
|                   {fprintf(logic_out, "policy_action('%s', deny).\n", (policy_id)->name);
|                   if (sym_look ((policy_id)->name, TARGET_ALL) == FOUND_ENTRY)
|                   {fprintf(logic_out, "target('%s', deny_all).\n", (policy_id)->name);}}
|                   action
|                   action_list ',' action
;

action:              action_reserved_word

```

```

|         action_reserved_word ASSIGN number_units
;

action_reserved_word: PRIORITY
    {fprintf(logic_out,"policy_action('%s',permit).\n",(policy_id)->name);
    if (sym_look ((policy_id)->name,
        TARGET_ALL) == FOUND_ENTRY)
    {fprintf(logic_out,"target('%s',permit_all).\n",(policy_id)->name);}};
|     PERMIT
    {fprintf(logic_out,"policy_action('%s',permit).\n",(policy_id)->name);
    if (sym_look ((policy_id)->name,
        TARGET_ALL) == FOUND_ENTRY)
        {fprintf(logic_out,"target('%s',permit_all).\n",
            (policy_id)->name);}};
|     HOPCOUNT
    {fprintf(logic_out,"policy_action('%s',permit).\n",(policy_id)->name);
    if (sym_look ((policy_id)->name,
        TARGET_ALL) == FOUND_ENTRY)
    {fprintf(logic_out,"target('%s',permit_all).\n",(policy_id)->name);}};
;

%%

extern FILE *yyin;
extern FILE *logic_out;

main(int ac, char **av)
{
    FILE *prologrules;      /* Input files of ProLog policy rules */
    int i,j,count,c,first_time; /* Loop counters and flags */
    /*
    * File to contain Prolog Facts from PPL Configuration File
    * and the ProLog policy rules
    */
    char *logic_file_name;

    /*
    * Scan the arguments passed in with the compile statement.
    * Set the appropriate flags according to the user's request
    */
    while (--ac > 0){
        /* User wanted the Symbol Tables Dumped after parse */
        if (strcmp (av[ac],"-v") == 0){
            verbose = 1;
        }
        /* User requests no wild card expansion */
        else if (strcmp (av[ac],"-no_wild") == 0){

```

```

        no_wild = 1;
    }
    /*
    * Do not implicit denies in the conflict detection
    * phase if the policy's creator are the same.
    */
    else if (strcmp (av[ac], "-no_implicit_deny") == 0){
        no_implicit_deny = 1;
    }
    /* Test the input configuration file */
    else {
        if ((yyin = fopen(av[ac], "r")) == NULL) {
            perror (av[ac]);
            exit(1);
        }
        logic_file_name = (char *)malloc ((size_t)15);
        sprintf (logic_file_name, "ppl1.txt");

        if ((logic_out = fopen(logic_file_name, "w")) == NULL) {
            perror (logic_file_name);
            exit(1);
        }
    }
}

/*
* If the input file was not specified or invalid,
* print compiler usage to the user
*/
if (yyin == NULL){
    print_usage();
    exit(1);
}

/* Provide parsing output and number of policies parsed */
if (!yyvsparse()){
    printf ("\nParse worked\n");
    if (policy_count == 0){
        printf ("No policies specified\n\n");
        exit(0);
    }
    else
        printf ("%d policies scanned\n\n", policy_count);
}
else
    printf ("\nParse failed\n\n");

```

```

/*
 * Loop thru the symbol table to create ProLog facts about
 * the nodes used to construct the network.
 */
count = 0;
for (i = 0; i < MAXSYMBOLS; i++){
    if (symtable[i].name == NULL)
        break;
    switch (symtable[i].type) {
        case DEFINED_NODE:
            fprintf (logic_out, "node_label('%s',%i).\n",
                    symtable[i].name, count++);
            break;
    }
}

/*
 * Loop thru the symbol table to create ProLog facts about
 * user defined classes of traffic.
 */
count = 0;
for (i = 0; i < MAXSYMBOLS; i++){
    if (symtable[i].name == NULL)
        break;
    switch (symtable[i].type) {
        case DEFINED_CLASS:
            first_time = 1;
            fprintf (logic_out, "class('%s',[" , symtable[i].name);
            for (j = 0; j < MAXCLASSENTRIES; j++){
                if (class_entries[j].name == NULL)
                    break;
                if (class_entries[j].class_reference == &(symtable[i])){
                    if (first_time){
                        fprintf (logic_out, ""%s", class_entries[j].name);
                        first_time = 0;
                    }
                    else
                        fprintf (logic_out, ","%s", class_entries[j].name);
                }
            }
            fprintf (logic_out, "]).\n");
            break;
    }
}

/*

```

```

* Loop thru the symbol table to create ProLog facts about
* user defined types to be used in conditional statements,
* and users defined who can create network policies.
*/
count = 0;
for (i = 0; i < MAXSYMBOLS; i++){
    if (symtable[i].name == NULL)
        break;
    switch (symtable[i].type) {
        case DEFINED_TYPE:
            for (j = 0; j < MAXCLASSENTRIES; j++){
                if (class_entries[j].name == NULL)
                    break;
                if (class_entries[j].class_reference == &(symtable[i])){
                    fprintf (logic_out, "type('%s', '%s', %d).\n",
                        symtable[i].name, class_entries[j].name, class_entries[j].value);
                }
            }
            break;
        case DEFINED_USER:
            for (j = 0; j < MAXCLASSENTRIES; j++){
                if (class_entries[j].name == NULL)
                    break;
                if (class_entries[j].class_reference == &(symtable[i])){
                    fprintf (logic_out, "user('%s', '%s').\n",
                        symtable[i].name, class_entries[j].name);
                }
            }
            break;
    }
}

/*
* Output dummy facts to keep ProLog interpreter from choking when
* referencing facts that do not exist.
*/
fprintf (logic_out, "no_conditions('\_null\_').\n");
fprintf (logic_out, "target('\_null\_', '\_null\_').\n");
fprintf (logic_out, "path_message('\_null\_', '\_null\_').\n");
fprintf (logic_out, "policy_message('\_null\_', '\_null\_').\n");
fprintf (logic_out, "condition_path('\_null\_', '\_null\_', '\_null\_', '\_null\_').\n");

/*
* Set flag in Prolog to allow wild card characters to be expanded
* or not, depending on the user's direction
*/

```

```

if (no_wild)
    fprintf (logic_out, "wild(\no\').\n");
else
    fprintf (logic_out, "wild(\yes\').\n");

/*
 * Set flag in Prolog to allow implicit denies between
 * policies created by the same user to be ignored.
 */
if (no_implicit_deny)
    fprintf (logic_out, "user_implicit_deny(\no\').\n");
else
    fprintf (logic_out, "user_implicit_deny(\yes\').\n");

/*
 * Create and run the first stage of the conflict detection
 * process. This involves sorting the facts just created by
 * the parsing process. The sorting is needed only to stop
 * Prolog from issuing warnings about facts not being contiguous
 * in the file. Add to the end of the file, the Prolog rules that
 * are to be applied to the facts entered from the prologrules1.txt file.
 * Then execute the Prolog program by calling the Prolog command
 * line interface with input file.
 */
fclose(logic_out);
system("sort ppl1.txt > ppl1.pl");

if ((logic_out = fopen("ppl1.pl", "a")) == NULL) {
    perror ("ppl1.pl");
    exit(1);
}

if ((prologrules = fopen("prologrules1.txt", "r")) == NULL) {
    perror (av[1]);
    exit(1);
}

while ((c = getc(prologrules)) != EOF)
    putc(c, logic_out);

fclose(prologrules);
fclose(logic_out);

/* If the user asked for verbose mode, dump the symbol tables. */
if (verbose){
    dump_tables();
}

```

```

system("pl/bin/plcon.exe -f ppl1.pl -t stage1");

/*
 * Stage two of the conflict detection takes the modified facts
 * generated by stage1, and adding the rules to be applied together
 * in a file named ppl2.pl.
 * The Prolog interpreter then executes this next stage. Stage two like
 * stage one manipulates the facts from stage to stage in a progressive
 * fashion. One stage could have been used, but would be complicated to
 * follow and debug.
 * For efficiency, the combining of stages should be considered.
 */
system("sort ppl2.txt > ppl2.pl");
if ((logic_out = fopen("ppl2.pl", "a")) == NULL) {
    perror("ppl2.pl");
    exit(1);
}
if ((prologrules = fopen("prologrules2.txt", "r")) == NULL) {
    perror(av[1]);
    exit(1);
}

while ((c = getc(prologrules)) != EOF)
    putc(c, logic_out);
fclose(prologrules);

fclose(logic_out);

system("pl/bin/plcon.exe -f ppl2.pl -t stage2");

/*
 * Stage three is the final stage of the conflict detection and
 * resolution phase of the compiler. The modified facts from
 * stage two are used and applied with the Prolog rules from
 * the file "prologrules3.txt".
 * Final output is created in "scan_out.txt".
 */

system("sort ppl3.txt > ppl3.pl");

if ((logic_out = fopen("ppl3.pl", "a")) == NULL) {
    perror("ppl3.pl");
    exit(1);
}
if ((prologrules = fopen("prologrules3.txt", "r")) == NULL) {
    perror(av[1]);
    exit(1);
}

```



```

    }

    while ((c = getc(prologrules)) != EOF)
        putc(c, logic_out);

    fclose(prologrules);
    fclose(logic_out);

    system("pl/bin/plcon.exe -f ppl3.pl -t stage3");

    printf("\n\n Policy Conflict results written to \"scan_out.txt\"\n\n");
}

/*
 * Print_usage outputs the options available to the user trying to compile
 * PPL configuration file.
 */
void print_usage()
{
    printf("\n\n Usage: Scan [-v -wild -no_implicit_deny] <filename>\n");
    printf("\t -v          : Verbose mode, dump symbol tables\n");
    printf("\t -no_wild       : Do not support wild cards in paths\n");
    printf("\t -no_implicit_deny : If policy creators are the same between\n");
    printf("\t                  a conflict, do not enforce implicit deny\n");
    printf("\t filename       : Filename of PPL configuration file\n");
    printf("\n\n");
}

```

APPENDIX E. PROLOG CONFLICT DETECTION CODE

```
% Stage1 is the first step in determining policy conflicts.
% Steps: * Open "ppl2.txt" file for policy conflict information
%        * Print all the possible paths thru the network,
%        This is needed for matching paths represented with
%        wild cards.
%        * Print the association between policy labels and
%        policy targets.
%        * Print the association between policy labels and
%        policy conditions.
%        * Copy all other necessary facts from stage one to
%        the file that will be used in stage two.
%        * Close the output file
%
stagel:-
    %
    % Open the file ppl2.txt to place the modified ProLog facts
    % to be used in stage two.
    %
    open('ppl2.txt',write,output),
    set_output(output),

    %
    % Output paths & all the possible nodes that can be used
    % to create the path.
    %
    setof(Policy_Path,paths_in_nodes(Policy_Path),Policy_Paths),
    not(print_policy_list(Policy_Paths)),

    %
    % Create an association between a policy and the targets
    % it supports.
    %
    not(create_paths),!,
    setof(Target_List,policy_target_list(Target_List),Target_Lists),
    not(print_target_list(Target_Lists)),

    %
    % Create a list of conditions and associate them with policies
    %
    not(print_condition_list),

    %
```

```

% Forward the facts of users, bandwidth, no_conditions, actions and types
% to the file for stage two.
%
not(explicit_nodes),
not(print_no_conditions),
print_implicit_deny,
not(print_path_params),
not(print_users),
not(print_actions),
not(print_types),
not(print_target_all),
not(print_policy_owner),
not(print_nodes),
not(print_links),
not(print_path_messages),
not(print_policy_messages),

%
% Close the output file
%
set_output(user_output),
close(output),
write('Stage 1 complete'),nl,nl.

%=====
% Generate the paths required to detect policy conflicts
% If no wild card characters were used:
%     then print just the paths explicitly listed
%     and all the links that create the network
%     else print out all possible paths through the
%     network so that wild card matching can be done
%
create_paths:-
    wild('no'),
    not(explicit_paths),!,
    not(explicit_links),!,fail.

create_paths:-
    wild('yes'),
    findall(Node,node_label(Node,_),Nodes),!,
    create_paths(Nodes),!,fail.

%=====
% Return all possible paths incrementally by
% taking each node pair in the network and
% generating all possible paths between those

```

```

% to nodes.
%
create_paths([]):-fail,!.

create_paths([_|[]]):-fail,!.

create_paths([Node1,Node2|Tail]):-
    not(create_paths_helper([Node1,Node2|Tail])),!,
    not(create_paths([Node2|Tail])),!.

%=====
% Find all possible paths between two nodes
% in the network
%
create_paths_helper([]):-fail,!.

create_paths_helper([_|[]]):-fail,!.

create_paths_helper([Node1,Node2|Tail]):-
    not(all_paths(Node1,Node2)),!,
    not(all_paths(Node2,Node1)),!,
    not(create_paths_helper([Node1|Tail])),fail,!.

%=====
% Print all paths explicitly listed by the user
%
explicit_paths:- setof(Path, path_links(_, Path), Paths),
    print_path_list(Paths),fail.

%=====
% Print all possible links in the network
%
explicit_links:- findall(Link, link(Link,_,_), Links),!,
    remove_dups(Links, Links_nodup),!,
    print_link_list(Links_nodup),!,fail.

%=====
% Print all the nodes in the network
%
explicit_nodes:- setof(Node, node_label(Node,_), Nodes),
    print_node_list(Nodes),fail.

%=====
% Print out nicely all the possible paths of a network
% of length 2 or more. The atom's are quoted.

```

```

all_paths(A, Z) :- setof(Path, path1(A, [Z], Path), Paths),
                  print_path_list(Paths),fail.

```

```

%=====
% Helper function to "all_paths". Determines
% if two nodes in the network are directly connected.
%
path1(A, [A | Path1], [A | Path1]).

```

```

path1(A, [Y|Path1], Path) :- adjacent(X, Y),
                               not(member(X, Path1)),
                               path1(A, [X, Y | Path1], Path).

```

```

%=====
% Print out the paths associated with each policy
%
paths:- setof(Path,paths_in_nodes(Path),Paths),
        qsort(Paths,Sorted),
        not(print_list(Sorted)).

```

```

%=====
% Print out a node list nicely formatted
%
write_path([]):- write(']').

```

```

write_path([X|[]]):- term_to_atom(X,X_atom),
                    write(X_atom),
                    write(']'),!,true.

```

```

write_path([X|Tail]):- term_to_atom(X,X_atom),
                      write(X_atom),
                      write(','),
                      write_path(Tail).

```

```

%=====
% Print out a path represented as a list of nodes
%
print_node_list([]):-fail.

```

```

print_node_list([Link|Tail]):-
    node_label(Link,_),
    write('path(['),
    term_to_atom(Link,Link_atom),
    write(Link_atom),
    write(']').),nl,
    print_node_list(Tail),fail.

```

```

%=====
% Print out all the links in the network
% nicely formatted.
%
print_link_list([]):-fail.

print_link_list([Link|Tail]):-
    link(Link,Src,Dst),!,
    term_to_atom(Src,Src_atom),
    term_to_atom(Dst,Dst_atom),
    write('path(['),
    write(Src_atom),
    write(','),
    write(Dst_atom),
    write(']').'),nl,
    write('path(['),
    write(Dst_atom),
    write(','),
    write(Src_atom),
    write(']').'),nl,
    print_link_list(Tail),!,fail.

%=====
% print_path_list:
% Output a list of lists, where each
% list is on a line by itself and
% all the atoms are quoted.

print_path_list([]):- fail.

print_path_list([Path | Tail]) :-
    not(eq1(Path)),
    write('path(['),
    write_path(Path),
    write(')'), nl, print_path_list(Tail).

print_path_list([[X|Y] | Tail]) :-
    eq1([X|Y]),
    print_path_list(Tail).

print_path_list([[X|Y] | Tail]) :-
    eq1([X|Y]),
    node_label(X,_),
    write('path(['),
    term_to_atom(X,X_atom),
    write(X_atom),
    write(']').'),nl,

```

```

    print_path_list(Tail).

%=====
% print_policy_list:
% Output nicely for each policy the paths
% associated with the that policy

print_policy_list([]):- fail.
print_policy_list([[Label|Path]] | Tail) :-
    write('policy_path('),
    term_to_atom(Label,Label_atom),
    write(Label_atom),
    write(','),
    write_path(Path),
    write(').'), nl, print_policy_list(Tail).

%=====
% Print out a list of targets associated with a policy
%
write_target_path([X|[]]):- term_to_atom(X,X_atom),
    write(X_atom),!,true.

write_target_path([X|Tail]):- term_to_atom(X,X_atom),
    write(X_atom),
    write(','),
    write_target_path(Tail).

%=====
% Print out the association between a policy
% and its targets
%
print_target_list([[Label|Targets] | Tail]):-
    write('policy_target('),
    term_to_atom(Label,Label_atom),
    write(Label_atom),
    write(','),
    write_target_path(Targets),
    write(').'), nl,
    print_target_list(Tail).

%=====
% Given a Policy_Label a list will be returned
% with the first element being the Label, and
% the second element being the path associated
% with that label
% ex: ['Policy1',['NPS','IETF']]

```

```

paths_in_nodes([Policy_Label|[Expanded_Path]]):-
    path_links(Path,Path_List),
    policy_path(Policy_Label,Path),
    path_in_nodes(Path_List,Expanded_Path).

```

```

paths_in_nodes([Policy_Label|[Expanded_Path]]):-
    link(Path,_,_),
    policy_path(Policy_Label,Path),
    path_in_nodes([Path],Expanded_Path).

```

```

paths_in_nodes([Policy_Label|[Expanded_Path]]):-
    node_label(Path,_),
    policy_path(Policy_Label,Path),
    path_in_nodes([Path],Expanded_Path).

```

```

%=====
% Take a path with link elements, and return a path
% with those links expanded into its node
% components: wildcard characters are left
% untouched.
%
% Given: ['NPS_DARPA','DARPA_SPAWAR']
% Returns: ['NPS', 'DARPA', 'SPAWAR']

```

```

path_in_nodes([X],[N1,N2]):- link(X,N1,N2).

```

```

path_in_nodes([X],[X]):- node_label(X,_).

```

```

path_in_nodes([X],[X]):- X = '*'.

```

```

path_in_nodes([X|Tail],[X|Path]):-
    node_label(X,_),
    path_in_nodes(Tail,Path).

```

```

path_in_nodes([X|Tail],[X|Path]):-
    X = '*',
    path_in_nodes(Tail,Path).

```

```

%=====
%Print a list.
%
print_list([]):- fail.
print_list([X | Tail]) :- write(X), nl, print_list(Tail).

```

```

%=====
% Use quicksort to sort a list of lists by the

```



```

% number of elements in each list
qsort( [], []).

qsort([X | Tail], Sorted) :-
    split( X, Tail, Small, Big),
    qsort( Small, SortedSmall),
    qsort( Big, SortedBig),
    conc( SortedSmall, [X | SortedBig], Sorted).

%=====
% Helper function to qsort, splits one list
% into two parts
%
split( _, [], [], []).

split( X, [Y | Tail], [Y | Small], Big) :-
    gt( X, Y),!,
    split(X, Tail, Small, Big).

split(X, [Y | Tail], Small, [Y | Big]) :-
    split(X, Tail, Small, Big).

%=====
% Is X is greater than Y
%
gt(X, Y) :- length(X, Xlen) , length(Y, Ylen), Xlen > Ylen.

%=====
% Return the first element of a list
%
first([], []).
first( X, [X | _ ]).

%=====
% Given list of traffic classes, expand list
% by assigning action and operation, to
% each class in the list.
%
expand_value_list(_, Action, Class, Op, List, [Expanded_List]) :-
    expand_list(Action, Class, Op, List, Expanded_List).

%=====
% Helper to "expand_value_list"
% Assign action, and operation to each value
% defined for the traffic class.
%
expand_list( _, _, _, [], []).

```

```

expand_list(Action,Class,Op,[X|Tail],[Action,Class,Op,X|Results]):-
    expand_list(Action,Class,Op,Tail,Results).

```

```

%=====
% Remove duplicates from a list
%
remove_dups([],[]).

```

```

remove_dups([Head|Tail],List):-
    member(Head,Tail),
    remove_dups(Tail,List).

```

```

remove_dups([Head|Tail],[Head|List]):-
    not(member(Head,Tail)),
    remove_dups(Tail,List).

```

```

%=====
% Create a list of targets associated with a policy
%
policy_target_list([Label|[Results]]):-
    policy_targets(Label,Targets),
    create_target_list(Label,Targets,Target_list),
    remove_dups(Target_list,Unique_Target_List),
    flatten(Unique_Target_List, Results).

```

```

%=====
% Given a label to identify a policy, create
% a list of target traffic classes that the
% policy applies to.
%
create_target_list(_,[],[]).

```

```

create_target_list(Label,[_,!|_Tail],Results_of_Tail):-
    policy_action(Label,'deny'),
    create_target_list(Label,Tail,Results_of_Tail).

```

```

create_target_list(Label,[_,*|_Tail],Results_of_Tail):-
    policy_action(Label,_),
    create_target_list(Label,Tail,Results_of_Tail).

```

```

create_target_list(Label,[Class,'=|_',Target_List|Tail],[[Results_of_expand]]Results_of_Tail):-
    policy_action(Label,Action),
    remove_dups(Target_List, Unique_Target_List),
    expand_value_list(Label,Action,Class,'=|_',Unique_Target_List,Results_of_expand),
    create_target_list(Label,Tail,Results_of_Tail).

```

```

%=====
% Create a list of conditions that must be met
% in order for the policy to be executed.
%
list_of_conditions(Policy,_,[Policy,permit,Attribute,Op,Value]):-
    condition_path(Policy,Attribute,Op,Value),
    not(Op = '!='),
    not(type(Attribute,_,_)).

list_of_conditions(Policy,_,[Policy,permit,Attribute,'!=',Value]):-
    condition_path(Policy,Attribute,'!=',Value),
    not(type(Attribute,_,_)).

%=====
% Create a list conditions that are composed
% of user defined types.
%
list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Value]):-
    condition_path(Policy,Type,'==',Value).

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Type_Element]):-
    condition_path(Policy,Type,'!=',_),
    type(Type,Type_Element,_),
    setof(Values,condition_path(Policy,Type,'!=',Values),Value_Set),
    not(member(Type_Element,Value_Set)).

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Element2]):-
    condition_path(Policy,Type,'<=',Element),
    type(Type,Element,Value),
    type(Type,Element2,Value2),
    Value2 <= Value.

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Element2]):-
    condition_path(Policy,Type,'>=',Element),
    type(Type,Element,Value),
    type(Type,Element2,Value2),
    Value2 >= Value.

%=====
% Print the conditions that must be meet to
% execute the policy
%
output_conditions([]):-fail.

output_conditions([[Policy,Action,Attribute,Operator,Value]|Tail]):-
    not(Attribute = 'BW'),
    write('policy_condition('),

```

```

term_to_atom(Policy,A_Policy),write(A_Policy),
write(','),
write(Action),
write(','),
term_to_atom(Attribute,A_Attribute),write(A_Attribute),
write(','),
term_to_atom(Operator,A_Operator),write(A_Operator),
write(','),
term_to_atom(Value,A_Value),write(A_Value),
write(').'),nl,
output_conditions(Tail).

output_conditions([[Policy,Action,Attribute,Operator,Value]|Tail]):-
    Attribute = 'BW',
    write('policy_condition('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    write(Action),
    write(','),
    term_to_atom(Attribute,A_Attribute),write(A_Attribute),
    write(','),
    term_to_atom(Operator,A_Operator),write(A_Operator),
    write(','),
    convert_bw(Value,New_Value),
    write(New_Value),
    write(').'),nl,
    output_conditions(Tail).

%=====

output_no_conditions([]):-fail.

output_no_conditions([Policy|Tail]):-
    write('no_conditions('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(').'),nl,
    output_no_conditions(Tail).

%=====
% When no conditions are associated with a
% policy, make note of it.
%
print_no_conditions:-
    setof(Policy,no_conditions(Policy),No_Conditions),
    output_no_conditions(No_Conditions),
    true.

%=====

```

```

% Print out the fact if implicit denies are
% to applied to conflict detection when both
% policies are created by the same user
%
print_implicit_deny:-
    user_implicit_deny(Option),
    write('user_implicit_deny('),
    write(Option),
    write(').'),nl,
    true.

%=====
% Print out a user defined type
% Helper to "print_types"
%
output_types([]):-fail.

output_types([[Type,Element,Value]|Tail]):-
    write('type('),
    term_to_atom(Type,A_Type),write(A_Type),
    write(','),
    term_to_atom(Element,A_Element),write(A_Element),
    write(','),
    write(Value),
    write(').'),nl,
    output_types(Tail).

%=====
% Print out all user defined types
%
print_types:-
    setof([Type,Element,Value],type(Type,Element,Value),Types),
    output_types(Types),
    true.

%=====
% For each policy, print out the target classes
% effected by it.
%
print_target_all:-
    setof(Policy,target(Policy,_),Policies),
    output_target_all(Policies),
    true.

%=====
% Print out the class of traffic effected by a policy
%
output_target_all([]):-fail.

```

```

output_target_all([Policy|Tail):-
    target(Policy,Value),
    write('target('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_target_all(Tail).

```

```

%=====
% Print out facts about the creator/owner of
% each policy to be applied to the network.
%
print_policy_owner:-
    setof(Policy,policy_owner(Policy,_),Policies),
    output_policy_owner(Policies),
    true.

```

```

%=====
% Output the owner for a policy
%
output_policy_owner([]):-fail.

```

```

output_policy_owner([Policy|Tail):-
    policy_owner(Policy,Value),
    write('policy_owner('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_policy_owner(Tail).

```

```

%=====
% Print out all the nodes of the network.
%
print_nodes:-
    setof(Label,node_label(Label,_),Labels),
    output_nodes(Labels),
    true.

```

```

%=====
% Output a fact for each node in the network
% These facts are used in the conflict
% decision process.
%
output_nodes([]):-fail.

```

```

output_nodes([Label|Tail]):-
    node_label(Label,Value),
    write('node_label('),
    term_to_atom(Label,A_Label),write(A_Label),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_nodes(Tail).

%=====
% Print out all the "messages" associated with a
% link.
%
print_path_messages:-
    setof(Link,path_message(Link,_),Links),
    output_path_messages(Links),
    true.

%=====
% Print the messages associated with a policy
%
print_policy_messages:-
    setof(Policy,policy_message(Policy,_),Policies),
    output_policy_messages(Policies),
    true.

%=====
% Output the "message" associated with a path
%
output_path_messages([]):-fail.

output_path_messages([Link|Tail]):-
    path_message(Link,Message),
    write('path_message('),
    term_to_atom(Link,A_Link),write(A_Link),
    write(','),
    term_to_atom(Message,A_Message),write(A_Message),
    write(').'),nl,
    output_path_messages(Tail).

%=====
% Output the "message" associated with a policy
%
output_policy_messages([]):-fail.

output_policy_messages([Policy|Tail]):-
    policy_message(Policy,Message),
    write('policy_message('),
    term_to_atom(Policy,A_Policy),write(A_Policy),

```

```

write(','),
term_to_atom(Message,A_Message),write(A_Message),
write(').'),nl,
output_policy_messages(Tail).

```

```

% Print out all the links of the network
%
print_links:-
    findall(Link,link(Link,_,_),Links),!,
    remove_dups(Links, Links_nodup),!,
    output_links(Links_nodup),!,
    true.

```

```

% Output a link of the network
%
output_links([]):-fail.

```

```

output_links([Link|Tail]):-
    link(Link,Src,Dst),!,
    term_to_atom(Src,Src_atom),
    term_to_atom(Dst,Dst_atom),
    term_to_atom(Link,Link_atom),
    write('link('),
    write(Link_atom),
    write(','),
    write(Src_atom),
    write(','),
    write(Dst_atom),
    write(').'),nl,
    write('link('),
    write(Link_atom),
    write(','),
    write(Dst_atom),
    write(','),
    write(Src_atom),
    write(').'),nl,
    output_links(Tail),!,fail.

```

```

% Output a user that is allowed to create
% policies.
%
output_users([]):-fail.

```

```

output_users([User|Tail]):-

```



```

        user(User,Level),
        write('user('),
        term_to_atom(User,A_User),write(A_User),
        write(','),
        term_to_atom(Level,A_Level),write(A_Level),
        write(').'),nl,
        output_users(Tail).

%=====
% Output all the actions associated with each policy
%
output_actions([]):-fail.

output_actions([Policy|Tail]):-
    policy_action(Policy,Action),
    write('policy_action('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Action,A_Action),write(A_Action),
    write(').'),nl,
    output_actions(Tail).

%=====
% Print all the users that are allowed to
% create policies.
%
print_users:-
    setof(User,user(User,_), User_list),
    output_users(User_list),
    true.

%=====
print_actions:-
    setof(Policy,policy_action(Policy,_), Action_list),
    output_actions(Action_list),
    true.

%=====
% Output the parameters associated with a path
% this includes bandwidth.
%
output_params([]):-fail.

output_params([Path|Tail]):-
    path_param(Path,Att,Op,Value,Unit),
    write('path_param('),
    term_to_atom(Path,A_Path),write(A_Path),
    write(','),

```

```

term_to_atom(Att,A_Att),write(A_Att),
write(','),
term_to_atom(Op,A_Op),write(A_Op),
write(','),
convert_unit(Value,Unit,New_Value),
term_to_atom(New_Value,A_Value),write(A_Value),
write(','),
term_to_atom('MBPS',A_Unit),write(A_Unit),
write(').').nl,
output_params(Tail).

%=====
% Print out all the paramaters associated with
% each path
%
print_path_params:-
    setof(Path, path_param(Path,_,_,_),Paths),
    output_params(Paths),
    true.

%=====
% Print out the conditions of all policies
%
print_condition_list:-
    setof(Condition_Set,list_of_conditions(Condition_Set), Conditions),
    output_conditions(Conditions),
    true.

%=====
% Print out the user defined types involved in the
% conditions of all policies
%
print_condition_list:-
    setof(Condition_Set,list_of_type_conditions(Condition_Set), Conditions),
    output_conditions(Conditions),
    true.

%=====
% Convert Bandwidth unit to a uniform Mbps for
% comparison reasons
%
convert_bw([Value,Unit],New_Value):-
    convert_unit(Value,Unit,New_Value).

%=====
convert_unit(Old_Value,'MBPS',Old_Value).

```

```
convert_unit(Old_Value,'GBPS',New_Value):-  
    New_Value is Old_Value * 1024.
```

```
convert_unit(Old_Value,'KBPS',New_Value):-  
    New_Value is Old_Value / 1024.
```

```
convert_unit(Old_Value,'BPS',New_Value):-  
    New_Value is (Old_Value / 1024)/1024.
```

```
%=====
```

```
% True if there is an link from X->Y or Y->X (undirected link)
```

```
%
```

```
adjacent(X, Y) :- link(_X,Y); link(_Y,X).
```

```
%=====
```

```
% Concat two lists together
```

```
%
```

```
conc([],L,L).
```

```
conc( [X | L1], L2, [X | L3] ) :- conc(L1,L2,L3).
```

```
%=====
```

```
% Is the list of size 1?
```

```
%
```

```
eq1(_L[_]).
```

```

% Stage2 is the second step in determining policy conflicts.
% Steps: * Open "ppl3.txt" file for policy conflict information
%      * Print all paths in the next that are associated with
%      policies being applied to the network.
%      * Copy all other necessary facts from stage two to
%      the file that will be used in third and final stage three.
%      * Close the output file
%
stage2:- open('ppl3.txt',write,output),
        set_output(output),

        % From all possible paths in the network, print
        % only those associated with policies after the
        % expansion of all wild card characters.
        not(all_policy_paths),

        % Print all necessary facts for use in stage three
        not(print_no_conditions),
        not(print_path_params),
        not(print_users),
        print_implicit_deny,
        not(print_types),
        not(print_actions),
        not(print_target_all),
        not(print_policy_owner),
        not(print_nodes),
        not(print_links),
        not(print_path_messages),
        not(print_policy_messages),
        write('condition(_null,_null,_null,_null,_null,_null).'),nl,

        set_output(user_output),
        close(output),
        write('Stage 2 complete'),nl,nl.

%=====
% Print all paths associated with policies
%
all_policy_paths:-
    policy_path(Pol_Label,Pol_Path),
    path(Pos_Path),
    setof(Policy_Path,policy_paths(Pol_Label,Pol_Path,Pos_Path,Policy_Path),Policy_Paths
),
    not(print_path_list(Policy_Paths)),
    print_conditions(Policy_Paths).

%=====

```

```

% Match wild card paths, with their expanded paths
%
policy_paths(Policy_Label, Policy_Path, Possible_Path,[Policy_Label|Possible_Path]):-
    match(Policy_Path,Possible_Path).

%=====
% Does a given list with or without wildcard
% characters match another one?
%
match([X|Tail1],[X|Tail2]):- match(Tail1,Tail2).

match(['*',*|Tail],Path):- match(['*|Tail],Path).

match(['*',X|Tail1],[X|Tail2]):- match(Tail1,Tail2).

match(['*',X|Tail1],[Y|Tail2]):- not(X=Y), match(['*',X|Tail1],Tail2).

match([],[]).

match([_,_,_],[]):- fail.
match([_,_,_],[_]):- fail.

match(['*'],_).

%=====
% print_path_list:
% Output a list of lists, where each
% list is on a line by itself and
% all the atoms are quoted.
%
print_path_list([]):- fail.

print_path_list([[Policy_Label|Path] | Tail]) :-
    policy_target(Policy_Label,Policy_Targets),
    not(eq1([Policy_Label|Path])),
    write('path('),
    term_to_atom(Policy_Label,Policy_atom),
    write(Policy_atom),
    write(','),
    write_path(Path),
    write(','),
    write_path(Policy_Targets),
    write(')'), nl, not(print_path_list(Tail)),fail.

print_path_list([[Policy_Label|Path] | Tail]) :-
    not(policy_target(Policy_Label,_)),
    not(eq1([Policy_Label|Path])),

```

```

        write('path('),
        term_to_atom(Policy_Label,Policy_atom),
        write(Policy_atom),
        write(','),
        write_path(Path),
        write('),'). nl,not(print_path_list(Tail)),fail.

%=====
% Print the conditions associated with every policy
%
print_conditions([]):- fail.

print_conditions([[Policy_Label|Path] | Tail]) :-
    policy_condition(Policy_Label,Action,Att,Op,Value),
    not(eq1([Policy_Label|Path])),
    write('condition('),
    term_to_atom(Policy_Label,Policy_atom),write(Policy_atom),
    write(','),
    write_path(Path),
    write(','),
    term_to_atom(Action,Action_atom),write(Action_atom),
    write(','),
    term_to_atom(Att,Att_atom),write(Att_atom),
    write(','),
    term_to_atom(Op,Op_atom),write(Op_atom),
    write(','),
    term_to_atom(Value,Value_atom),write(Value_atom),
    write('),'). nl, print_path_list(Tail).

print_conditions([[Policy_Label|Path] | Tail]) :-
    not(policy_condition(Policy_Label,_,_,_)),
    not(eq1([Policy_Label|Path])),
    print_path_list(Tail).

%=====
% Helper function used by print_path_list to output
% a list with all the atoms quoted.

write_path([]):- write(']').

write_path([X|_]):- term_to_atom(X,X_atom),
                    write(X_atom),
                    write(']'),!,true.

write_path([X|Tail]):- term_to_atom(X,X_atom),
                      write(X_atom),
                      write(','),

```

```

        write_path(Tail).

%=====
% Is the list of size 1?
eq1([_|[]]).

%=====
% Print a user defined type
%
output_types([]):-fail.

output_types([[Type,Element,Value]|Tail]):-
    write('type('),
    term_to_atom(Type,A_Type),write(A_Type),
    write(','),
    term_to_atom(Element,A_Element),write(A_Element),
    write(','),
    write(Value),
    write(').').nl,
    output_types(Tail).

%=====
% Print all the user defined types
%
print_types:-
    setof([Type,Element,Value],type(Type,Element,Value),Types),
    output_types(Types),
    true.

%=====
% Print a policy that has no conditions
% associated with it.
%
output_no_conditions([]):-fail.

output_no_conditions([Policy|Tail]):-
    write('no_conditions('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(').').nl,
    output_no_conditions(Tail).

%=====
% Print all the policies that have no conditions
% associated with them.
%
print_no_conditions:-
    setof(Policy,no_conditions(Policy),No_Conditions),
    not(No_Conditions = []),

```

```

        output_no_conditions(No_Conditions),
        true.

%=====
% Print the owners of all the policies
%
print_policy_owner:-
    setof(Policy,policy_owner(Policy,_),Policies),
    output_policy_owner(Policies),
    true.

%=====
% Print the owner of a policy
%
output_policy_owner([]):-fail.

output_policy_owner([Policy|Tail]):-
    policy_owner(Policy,Value),
    write('policy_owner('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_policy_owner(Tail).

%=====
% Print the nodes of the network.
%
print_nodes:-
    setof(Label,node_label(Label,_),Labels),
    output_nodes(Labels),
    true.

%=====
% Print out a node of the network
%
output_nodes([]):-fail.

output_nodes([Label|Tail]):-
    node_label(Label,Value),
    write('node_label('),
    term_to_atom(Label,A_Label),write(A_Label),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_nodes(Tail).

%=====

```



```

% Print out all the links of the network
%
print_links:-
    findall(Link,link(Link,_,_),Links),!,
    remove_dups(Links, Links_nodup),!,
    output_links(Links_nodup),!,
    true.

%=====
% Print out a link in the network.
%
output_links([]):-fail.

output_links([Link|Tail]):-
    link(Link,Src,Dst),!,
    term_to_atom(Src,Src_atom),
    term_to_atom(Dst,Dst_atom),
    term_to_atom(Link,Link_atom),
    write('link('),
    write(Link_atom),
    write(','),
    write(Src_atom),
    write(','),
    write(Dst_atom),
    write(').').nl,
    write('link('),
    write(Link_atom),
    write(','),
    write(Dst_atom),
    write(','),
    write(Src_atom),
    write(').').nl,
    output_links(Tail),!,fail.

%=====
% Print out messages associated with each path
%
print_path_messages:-
    setof(Link,path_message(Link,_,_),Links),
    output_path_messages(Links),true,!.

%=====
% Print the messages associated with each policy
%
print_policy_messages:-
    setof(Policy,policy_message(Policy,_,_),Policies),
    output_policy_messages(Policies),true,!.

```

```

%=====
% Print the messages associated with a path
%
output_path_messages([]):-fail.

output_path_messages([Link|Tail]):-
    path_message(Link,Message),
    write('path_message('),
    term_to_atom(Link,A_Link),write(A_Link),
    write(','),
    term_to_atom(Message,A_Message),write(A_Message),
    write(').').nl,
    output_path_messages(Tail),!.

%=====
% Print the messages associated with a path
%
output_policy_messages([]):-fail.

output_policy_messages([Policy|Tail]):-
    policy_message(Policy,Message),
    write('policy_message('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Message,A_Message),write(A_Message),
    write(').').nl,
    output_policy_messages(Tail),!.

%=====
% Print out the users who can created policies
%
output_users([]):-fail.

output_users([User|Tail]):-
    user(User,Level),
    write('user('),
    term_to_atom(User,A_User),write(A_User),
    write(','),
    write(Level),
    write(').').nl,
    output_users(Tail).

%=====
% Print the actions associated with policy
%
output_actions([]):-fail.

output_actions([Policy|Tail]):-

```

```

    policy_action(Policy,Action),
    write('policy_action('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Action,A_Action),write(A_Action),
    write(').'),nl,
    output_actions(Tail).

%=====
% Print the paramaters associated with a path
%
output_params([]):-fail.

output_params([Path|Tail]):-
    path_param(Path,Att,Op,Value,Unit),
    write('path_param('),
    term_to_atom(Path,A_Path),write(A_Path),
    write(','),
    term_to_atom(Att,A_Att),write(A_Att),
    write(','),
    term_to_atom(Op,A_Op),write(A_Op),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(','),
    term_to_atom(Unit,A_Unit),write(A_Unit),
    write(').'),nl,
    output_params(Tail).

%=====
% Print the parameters of each path
%
print_path_params:-
    setof(Path, path_param(Path,_,_,_),Paths),
    output_params(Paths),
    true.

%=====
% Print the flag identifying whether implicit
% denies are to be ignored between policies
% created by the same user
%
print_implicit_deny:-
    user_implicit_deny(Option),
    write('user_implicit_deny('),
    write(Option),
    write(').'),nl,
    true.

```

```

%=====
% Print all the user who can create a policy
%
print_users:-
    setof(User,user(User,_), User_list),
    output_users(User_list),
    true.

%=====
% Print out all the actions for each policy
%
print_actions:-
    setof(Policy,policy_action(Policy,_), Action_list),
    output_actions(Action_list),
    true.

%=====
% Print the targets for each policy
%
print_target_all:-
    setof(Policy,target(Policy,_),Policies),
    output_target_all(Policies),
    true.

%=====
% Print the target for a policy
%
output_target_all([]):-fail.

output_target_all([Policy|Tail]):-
    target(Policy,Value),
    write('target('),
    term_to_atom(Policy,A_Policy),write(A_Policy),
    write(','),
    term_to_atom(Value,A_Value),write(A_Value),
    write(').'),nl,
    output_target_all(Tail).

%=====
% Remove duplicate items from a list
%
remove_dups([],[]).

remove_dups([Head|Tail],List):-
    member(Head,Tail),
    remove_dups(Tail,List).

```

```
remove_dups([Head|Tail],[Head|List]):-  
    not(member(Head,Tail)),  
    remove_dups(Tail,List).
```

```

% Stage3 is the final step in determining policy conflicts.
% Steps: * Open "scan_out.txt" file for policy conflict information
%       * Find all policies with overlapping paths
%       * Determine policies in conflict
%       * Print the policy conflicts that could not be resolved
%       * Print the policy conflicts that COULD be resolved
%       * Print out the policies that contain "message" conflicts
%       * Close the output file
%
stage3:- open('scan_out.txt',write,output),
        set_output(output),

        % Find all policies expressed over links and nodes
        % that are sub-paths of other policies.
        %
        find_all_subpaths(Subpaths),

        % Take the list of policies that have overlapping
        % paths and check for permit conflicts
        %
        permit_conflicts(Subpaths,Permit_Conflicts),

        % Print out policy conflicts that can not be
        % resolved using the "Id" of the creator
        %
        write('          Print Unresolved Conflicts'),nl,
        write('          ====='),nl,
        print_unresolved_conflict_list(Permit_Conflicts),

        % Print out the policy conflicts than CAN be resolved
        % using the "Id" of the policy creator
        %
        nl,nl,nl,write('          Print Resolved Conflicts'),nl,
        write('          ====='),nl,
        print_resolved_conflict_list(Permit_Conflicts),

        % Print out the policies that require "message" support on a
        % path, but all the links of the path do not support that "message"
        %
        nl,nl,nl,write('          Message Conflicts'),nl,
        write('          ====='),nl,nl,
        not(print_message_conflict_list),

        %
        set_output(user_output),
        close(output),

```

```

write('Stage 3 complete'),nl,nl.

%=====
%
% Return set of all policies that contain overlapping
% paths.
%
find_all_subpaths(Subpaths):-
    setof(Subpath, find_subpaths(Subpath),Subpaths).

find_all_subpaths([]):-
    not(setof(Subpath, find_subpaths(Subpath),_)).

%=====
%
% Return all Policy/Path pairs were a policy specified
% over on a link or node, is a sublist of policies
% based on either nodes, links, or user defined paths.
%
find_subpaths([Policy1,Path1,Target1,Policy2,Path2,Target2]):-
    path(Policy1,Path1,Target1),
    link(_,From,To),
    [From,To] = Path1,
    path(Policy2,Path2,Target2),
    sublist(Path1,Path2).

find_subpaths([Policy1,Path1,Target1,Policy2,Path2,Target2]):-
    path(Policy1,Path1,Target1),
    node_label(Label,_),
    [Label] = Path1,
    path(Policy2,Path2,Target2),
    sublist(Path1,Path2).

%=====
% Take a list of policy pairs that contain overlapping paths.
% Check each policy pair for overlapping conditions.
% If the conditions do overlap, then check the classes of
% of traffic that are permitted on each to determine
% if a conflict exists.
%
permit_conflicts([[Policy1,Path1,Target1,Policy2,Path2,Target2]|Tail],
    [[Policy1,Path1,Target1,Policy2,Path2,Target2,Conflicts]|Permit_Conflicts]):-
    not(Policy1 = Policy2),
    conditional_overlap(Policy1,Policy2,_,No_Overlap),
    No_Overlap = [],
    setof(Conflict,conflict_permit_targets(Policy1,Policy2,Target2,Target1,Conflict),Conflic
ts),

```

```

not(Conflicts = []),
permit_conflicts(Tail, Permit_Conflicts).

permit_conflicts([[Policy1, Path1, Target1, Policy2, Path2, Target2]|Tail],
    [[Policy1, Path1, Target1, Policy2, Path2, Target2, [[Policy1, Value]]]|Permit_Conflicts]]:-
    not(Policy1 = Policy2),
    conditional_overlap(Policy1, Policy2, _, No_Overlap),
    No_Overlap = [],
    (
        (target(Policy1, permit_all),
         policy_action(Policy2, deny));
        (target(Policy1, deny_all),
         policy_action(Policy2, permit))
    ),
    target(Policy1, Value),
    permit_conflicts(Tail, Permit_Conflicts).

permit_conflicts([[Policy1, Path1, Target1, Policy2, Path2, Target2]|Tail],
    [[Policy1, Path1, Target1, Policy2, Path2, Target2, [[Policy2, Value]]]|Permit_Conflicts]]:-
    not(Policy1 = Policy2),
    conditional_overlap(Policy1, Policy2, _, No_Overlap),
    No_Overlap = [],
    (
        (target(Policy2, permit_all),
         policy_action(Policy1, deny));
        (target(Policy2, deny_all),
         policy_action(Policy1, permit))
    ),
    target(Policy2, Value),
    permit_conflicts(Tail, Permit_Conflicts).

%
% If there are no conflicts between two paths, then skip and continue to check
% other over laping paths.
%
permit_conflicts([[Policy1, _, Target1, Policy2, _, Target2]|Tail], Permit_Conflicts):-
    not(Policy1 = Policy2),
    setof(Conflict, conflict_permit_targets(Policy1, Policy2, Target2, Target1, Conflict), Conflic
ts),
    Conflicts = [],
    permit_conflicts(Tail, Permit_Conflicts).

%
% If the same policy is being checked, skip it and check the rest of the list
% of over laping paths.
%
permit_conflicts([[__, __, _]|Tail], Permit_Conflicts):-

```



```

    permit_conflicts(Tail,Permit_Conflicts).

%
% If no sub-paths, then no conflicts
%
permit_conflicts([],[]).

%=====
% Compare a target class element for conflicts with all
% the target elements of a second policy
%
conflict_permit_targets(,_,[],_).

conflict_permit_targets(Policy1,Policy2,[A1,C,_V_|_],Targets,Results):-
    conflict_permit_target(Policy1,Policy2,[A1,C,V],Targets,Results),
    not(empty(Results)).

conflict_permit_targets(Policy1,Policy2,[_,_,_Tail],Targets,Results):-
    conflict_permit_targets(Policy1,Policy2,Tail,Targets,Results),
    not(empty(Results)).

%=====
% Compare the target class and action between two
% two target elements.
%
conflict_permit_target(Policy1,Policy2,['permit',C,V],_,[C,V]):-
    policy_owner(Policy1, Creator1),
    policy_owner(Policy2, Creator2),
    Creator1 = Creator2,
    user_implicit_deny('no'),
    fail.

conflict_permit_target(,_,['permit',C,V],_,[C,V]):-
    user_implicit_deny('yes').

conflict_permit_target(,_,['deny',_,_],_,[]).

conflict_permit_target(_,_,_,[]).

conflict_permit_target(_,_,[A1,C,Value],[A2,C,_Value|_],[C,Value]):-
    not(A1 = A2),!.

conflict_permit_target(_,_,[A,C,Value],[A,C,_Value|_],[]) :-!.

conflict_permit_target(Policy1,Policy2,[A1,C1,Value1],[_,_,_Tail],Results):-
    conflict_permit_target(Policy1,Policy2,[A1,C1,Value1],Tail,Results).

```

```

% Determine if a conditional overlap exists by testing each
% type of condition.
%
conditional_overlap(P1,P2,['priority'|Over],No_Over):-
    priority_overlap(P1,P2),
    conditional_overlap1(P1,P2,Over,No_Over),!.

conditional_overlap(P1,P2,Over,['priority'|No_Over]):-
    not(priority_overlap(P1,P2)),
    conditional_overlap1(P1,P2,Over,No_Over),!.

conditional_overlap1(P1,P2,['time'|Over],No_Over):-
    time_overlap(P1,P2),
    conditional_overlap2(P1,P2,Over,No_Over),!.

conditional_overlap1(P1,P2,Over,['time'|No_Over]):-
    not(time_overlap(P1,P2)),
    conditional_overlap2(P1,P2,Over,No_Over),!.

conditional_overlap2(P1,P2,['hopcount'|Over],No_Over):-
    hopcount_overlap(P1,P2),
    conditional_overlap3(P1,P2,Over,No_Over),!.

conditional_overlap2(P1,P2,Over,['hopcount'|No_Over]):-
    not(hopcount_overlap(P1,P2)),
    conditional_overlap3(P1,P2,Over,No_Over),!.

conditional_overlap3(P1,P2,['bandwidth'|Over],No_Over):-
    bw_overlap(P1,P2),
    conditional_overlap4(P1,P2,Over,No_Over),!.

conditional_overlap3(P1,P2,Over,['bandwidth'|No_Over]):-
    not(bw_overlap(P1,P2)),
    conditional_overlap4(P1,P2,Over,No_Over),!.

conditional_overlap4(P1,P2,['user'|Over],No_Over):-
    user_overlap(P1,P2),
    conditional_overlap5(P1,P2,Over,No_Over),!.

conditional_overlap4(P1,P2,Over,['user'|No_Over]):-
    not(user_overlap(P1,P2)),
    conditional_overlap5(P1,P2,Over,No_Over),!.

conditional_overlap5(P1,P2,['host'|Over],No_Over):-
    host_overlap(P1,P2),
    conditional_overlap6(P1,P2,Over,No_Over),!.

```

```

conditional_overlap5(P1,P2,Over,['host'|No_Over]):-
    not(host_overlap(P1,P2)),
    conditional_overlap6(P1,P2,Over,No_Over),!.

conditional_overlap6(P1,P2,['type'|Over],No_Over):-
    not(overlap_types(P1,P2,_)),
    conditional_overlap7(P1,P2,Over,No_Over),!.

conditional_overlap6(P1,P2,Over,['type'|No_Over]):-
    overlap_types(P1,P2,_),
    conditional_overlap7(P1,P2,Over,No_Over),!.

conditional_overlap7(_,_,[],[]).

%=====
% Determine if there is an overlap between
% user defined types.
%
overlap_types(Policy1, Policy2, Type):-
    type(Type,_),
    setof(Element,condition(Policy1,_,_,Type,_,Element),Elements1),
    setof(Element,condition(Policy2,_,_,Type,_,Element),Elements2),
    not(Elements1 = []),
    not(Elements2 = []),
    intersection(Elements1,Elements2,I),
    I = [].

%=====
% Determine if a priority overlap exists
%
priority_overlap(Policy1,Policy1).

priority_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    (not(setof([Op1,Value1],condition(Policy1,_,_, 'priority',Op1,Value1),_));
    not(setof([Op2,Value2],condition(Policy2,_,_, 'priority',Op2,Value2),_))).

priority_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    setof([Op1,Value1],condition(Policy1,_,_, 'priority',Op1,Value1),Values1),
    setof([Op2,Value2],condition(Policy2,_,_, 'priority',Op2,Value2),Values2),
    overlap(Values1,Values2).

%=====
% Determine if a hopcount overlap exists
%
```

```
hopcount_overlap(Policy1,Policy1).
```

```
hopcount_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    (not(setof([Op1,Value1],condition(Policy1,_,_,hopcount,Op1,Value1),_));  
     not(setof([Op2,Value2],condition(Policy2,_,_,hopcount,Op2,Value2),_))).
```

```
hopcount_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    setof([Op1,Value1],condition(Policy1,_,_,hopcount,Op1,Value1),Values1),  
    setof([Op2,Value2],condition(Policy2,_,_,hopcount,Op2,Value2),Values2),  
    overlap(Values1,Values2).
```

```
%=====
```

```
% Determine if a bandwidth overlap exists
```

```
%
```

```
bw_overlap(Policy1,Policy1).
```

```
bw_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    (not(setof([Op1,Value1],condition(Policy1,_,_,BW,Op1,Value1),_));  
     not(setof([Op2,Value2],condition(Policy2,_,_,BW,Op2,Value2),_))).
```

```
bw_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    setof([Op1,Value1],condition(Policy1,_,_,BW,Op1,Value1),Values1),  
    setof([Op2,Value2],condition(Policy2,_,_,BW,Op2,Value2),Values2),  
    overlap(Values1,Values2).
```

```
%=====
```

```
% Determine if a time overlap exists
```

```
%
```

```
time_overlap(Policy1,Policy1).
```

```
time_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    (not(setof([Op1,Time1],condition(Policy1,_,_,time,Op1,Time1),_));  
     not(setof([Op2,Time2],condition(Policy2,_,_,time,Op2,Time2),_))).
```

```
time_overlap(Policy1,Policy2):-  
    not(Policy1 = Policy2),  
    setof([Op1,Time1],condition(Policy1,_,_,time,Op1,Time1),Times1),  
    setof([Op2,Time2],condition(Policy2,_,_,time,Op2,Time2),Times2),  
    overlap(Times1,Times2).
```

```
%=====
```

```
% An overlap exists if ALL elements of a condition
```

```
% overlap. The overlap rule uses the operator and value
% of two conditional elements to determine if overlaps
% exist
%
overlap([],[]).
```

```
overlap([],[]).
```

```
overlap([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2):-
    not([[Op2,Value2]|Tail2] = []),
    not([[Op1,Value1]|Tail1] = []),
    overlap1(Value1,Op1,Value2,Op2),
    overlap(Tail1,[Op2,Value2]|Tail2),
    overlap([[Op1,Value1]|Tail1],Tail2).
```

```
%=====
% Helper rule to "overlap" above.
% Determines if an overlap exists between
% two conditional elements.
%
overlap1(Value1,_,Value1,_).
```

```
overlap1(_,'>=','>=').
```

```
overlap1(Value1,'>=',Value2,'<='):-
    Value1 < Value2.
```

```
overlap1(_,'<=','<='):-true.
```

```
overlap1(Value1,'<=',Value2,'>='):-
    Value1 > Value2.
```

```
%=====
% Determines if there is an overlap between
% users. Each user in the conditional element
% list is checked, and if any user
% overlap exists, then the policy contains a
% user overlap.
%
user_overlap(Policy1,Policy1).
```

```
user_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    (not(setof([Op1,User1],condition(Policy1,_,_,_,'user_id',Op1,User1),_));
    not(setof([Op2,User2],condition(Policy2,_,_,_,'user_id',Op2,User2),_))).
```

```
user_overlap(Policy1,Policy2):-
```

```

    not(Policy1 = Policy2),
    setof([Op1,User1],condition(Policy1,_,_,user_id',Op1,User1),Users1),
    setof([Op2,User2],condition(Policy2,_,_,user_id',Op2,User2),Users2),
    user_overlap1(Users1,Users2,Results),
    flatten(Results,Flat_Results),
    not(member('conflict',Flat_Results)),
    member('overlap',Flat_Results),
    true.

user_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    setof([Op1,User1],condition(Policy1,_,_,user_id',Op1,User1),Users1),
    setof([Op2,User2],condition(Policy2,_,_,user_id',Op2,User2),Users2),
    user_overlap1(Users1,Users2,Results),
    flatten(Results,Flat_Results),
    member('conflict',Flat_Results),
    fail.

%=====
% Helper rule to "user_overlap" above.
% Determines if user element lists overlap
% with at least one common user
%
user_overlap1([],[],[]).

user_overlap1([_],[],[]).

user_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result1,Result2,Result3]):-
    not([Op2,Value2]|Tail2] = []),
    not([Op1,Value1]|Tail1] = []),
    user_overlap2(Value1,Op1,Value2,Op2,Result1),
    user_overlap1(Tail1,[Op2,Value2]|Tail2],Result2),
    user_overlap1([Op1,Value1]|Tail1,Tail2,Result3).

user_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
    not([Op2,Value2]|Tail2] = []),
    not([Op1,Value1]|Tail1] = []),
    not(user_overlap2(Value1,Op1,Value2,Op2,_)),
    user_overlap1(Tail1,[Op2,Value2]|Tail2],Result2),
    user_overlap1([Op1,Value1]|Tail1,Tail2,Result3).

%=====
% Helper rule to "user_overlap1" above
% Determines if two user values overlap
% Can return "No overlap", "Overlap", or "Conflict"
%
user_overlap2(Value,Op,Value,Op,'overlap').

```

```

user_overlap2(Value1,'==',Value2,'==','nooverlap'):-
    not(Value1 = Value2).

user_overlap2(Value1,'==',Value2,'!=','overlap'):-
    not(Value1 = Value2).

user_overlap2(Value1,'==',Value1,'!=','conflict').

% This makes an assumption that there are many/infinite users
% If there were limited user, then maybe they should be a "type" instead.
%
user_overlap2(,!=',,!=','overlap').

%=====
% Determines if there is an overlap in host identifiers
%
host_overlap(Policy1,Policy1).

host_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    (not(setof([Op1,Add1],condition(Policy1,_,_, 'host_id',Op1,Add1),_));
    not(setof([Op2,Add2],condition(Policy2,_,_, 'host_id',Op2,Add2),_))).

host_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    setof([Op1,Add1],condition(Policy1,_,_, 'host_id',Op1,Add1),Addresses1),
    setof([Op2,Add2],condition(Policy2,_,_, 'host_id',Op2,Add2),Addresses2),
    host_overlap1(Addresses1,Addresses2,Results),
    flatten(Results,Flat_Results),
    not(member('conflict',Flat_Results)),
    member('overlap',Flat_Results),
    true.

host_overlap(Policy1,Policy2):-
    not(Policy1 = Policy2),
    setof([Op1,Add1],condition(Policy1,_,_, 'host_id',Op1,Add1),Addresses1),
    setof([Op2,Add2],condition(Policy2,_,_, 'host_id',Op2,Add2),Addresses2),
    host_overlap1(Addresses1,Addresses2,Results),
    flatten(Results,Flat_Results),
    member('conflict',Flat_Results),
    fail.

%=====
% Helper rule to "host_overlap" above
% Progresses through the list of host addresses from
% two policies looking for an overlap

```

```

%
host_overlap1([],[],[]).

host_overlap1([_],[],[]).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result1,Result2,Result3]):-
    not([[Op2,Value2]|Tail2] = []),
    not([[Op1,Value1]|Tail1] = []),
    not(address_overlap(Value1,Value2)),
    host_overlap2(Op1,'X',Op2,'Y',Result1),
    host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
    host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result1,Result2,Result3]):-
    not([[Op2,Value2]|Tail2] = []),
    not([[Op1,Value1]|Tail1] = []),
    address_overlap(Value1,Value2),
    host_overlap2(Op1,'X',Op2,'X',Result1),
    host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
    host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
    not([[Op2,Value2]|Tail2] = []),
    not([[Op1,Value1]|Tail1] = []),
    address_overlap(Value1,Value2),
    not(host_overlap2(Op1,'X',Op2,'X',_)),
    host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
    host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
    not([[Op2,Value2]|Tail2] = []),
    not([[Op1,Value1]|Tail1] = []),
    not(address_overlap(Value1,Value2)),
    not(host_overlap2(Op1,'X',Op2,'Y',_)),
    host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
    host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

%=====
% Helper rule to "host_overlap1" above
% Determines if two host values overlap using
% the specified comparison operator
%
host_overlap2('==',X,'==',X,'overlap').

host_overlap2('==',X,'==',Y,'nooverlap'):- not(X = Y).

host_overlap2('==',X,'!=',X,'conflict').

```



```

host_overlap2('==',X,'!=',Y,'overlap'):- not(X = Y).

host_overlap2('!=',X,'==',X,'conflict').

host_overlap2('!=',X,'==',Y,'overlap'):- not(X = Y).

host_overlap2('!=',X,'!=',X,'overlap').

host_overlap2('!=',X,'!=',Y,'overlap'):- not(X = Y).

%=====
% Determines if IP protocol addresses overlap
%
address_overlap([Dot1|Tail1],[Dot2|Tail2]):-
    Dot1 = Dot2,
    address_overlap(Tail1,Tail2).

address_overlap([Dot1|_],[Dot2|_]):-
    Dot1 = '*';
    Dot2 = '*'.

address_overlap([],[]).

%=====
% Determines if message specified for a policy path
% is supported by each link that composes the path
%
message_conflict(Policy,[Policy,Results]):-
    path(Policy,_),
    setof(Link, message_conflict_helper(Policy,[_,Link]), Links),
    flatten(Links,Flattened_Links),
    generate_list_pairs(Flattened_Links, List_of_Pairs),
    remove_dups(List_of_Pairs,Results).

message_conflict(Policy,[]):-
    path(Policy,_),
    not(setof(Link, message_conflict_helper(Policy,[_,Link]), _)).

%=====
% Given a policy, return a list of links that do
% NOT support the needed "messages"
%
message_conflict_helper(Policy,[Policy,Results]):-
    path(Policy,Path,_),
    policy_message(Policy,Message),

```

```

        message_supported(Path,Message,Results),
        not(Results = []).

%=====
% Create a list of link, message pairs such that
% the pairs returned do NOT support the needed "message"
%
message_supported([],_,[]).

message_supported([Src,Dst],Message,[Link,Message]):-
    link(Link,Src,Dst),
    not(path_message(Link,Message)).

message_supported([Src,Dst],Message,[]):-
    link(Link,Src,Dst),
    path_message(Link,Message).

message_supported([Src,Dst|Tail], Message, Results):-
    link(Link,Src,Dst),
    path_message(Link,Message),
    message_supported([Dst|Tail], Message,Results).

message_supported([Src,Dst|Tail], Message, [Link,Message|Results]):-
    link(Link,Src,Dst),
    not(path_message(Link,Message)),
    message_supported([Dst|Tail], Message,Results).

%=====
% Modify a given list of items into a list of
% pairs, taking two elements at a time to form
% the pairs.
%
generate_list_pairs([],[]).

generate_list_pairs([Node,Message|Tail],[[Node,Message]|Tail_Results]):-
    generate_list_pairs(Tail,Tail_Results).

%=====
% Remove duplicate items from a list
%
remove_dups([],[]).

remove_dups([Head|Tail],List):-
    member(Head,Tail),
    remove_dups(Tail,List).

remove_dups([Head|Tail],[Head|List]):-
    not(member(Head,Tail)),

```

```

        remove_dups(Tail,List).
%=====
% Concat two lists together
%
conc([],L,L).
conc( [X | L1], L2, [X | L3] ) :- conc(L1,L2,L3).

%=====
% Is a list empty
%
empty([]):-true.
empty(_):-fail.

%=====
% Is S a sublist of L
%
sublist(S,L):-
    conc(_L2,L),
    conc(S,_L2).

%=====
% print_conflict_list:
% Output nicely each policy conflict that could not
% be resolved using the ID of policy's creator

print_unresolved_conflict_list([]).

print_unresolved_conflict_list([[_,_,_,_,_]|Tail):-
    print_unresolved_conflict_list(Tail).

print_unresolved_conflict_list([[Policy1,Path1,Target1,Policy2,Path2,Target2,Permit_results]|Tail):-
    policy_owner(Policy1,Owner1),
    policy_owner(Policy2,Owner2),
    user(Owner1,Value1),
    user(Owner2,Value2),
    Value1 = Value2,
    nl,nl,write('Conflict '),
    write(Policy1),
    write(' <=> '),
    write(Policy2),nl,
    write(' '),
    write(Policy1),
    write(' Path = '),
    write_path(Path1),nl,
    write(' '),
    write(Policy1),

```

```

write(' Targets: '),
write_targets(Target1),nl,
write(' '),
write(Policy2),
write(' Path = []'),
write_path(Path2),nl,
write(' '),
write(Policy2),
write(' Targets: '),
write_targets(Target2),nl,
write(' '),
write('Target Conflicts: '),
print_list(Permit_results),nl,
print_unresolved_conflict_list(Tail).

print_unresolved_conflict_list([[Policy1,_,_,Policy2,_,_,_] | Tail]) :-
    policy_owner(Policy1,Owner1),
    policy_owner(Policy2,Owner2),
    user(Owner1,Value1),
    user(Owner2,Value2),
    not(Value1 = Value2),
    print_unresolved_conflict_list(Tail).

%=====
% print_conflict_list:
% Output nicely policy conflicts that could be
% resolved using the ID of the policy's creator
%
print_resolved_conflict_list([]).

print_resolved_conflict_list([_,_,_,_,_] | Tail):-
    print_resolved_conflict_list(Tail).

print_resolved_conflict_list([[Policy1,Path1,Target1,Policy2,Path2,Target2,Permit_results] | Tail]
):-
    policy_owner(Policy1,Owner1),
    policy_owner(Policy2,Owner2),
    user(Owner1,Value1),
    user(Owner2,Value2),
    not(Value1 = Value2),
    nl,nl,write('Conflict '),
    write(Policy1),
    write(' <=> '),
    write(Policy2),nl,
    write(' '),
    write(Policy1),
    write(' Path = []'),

```

```

write_path(Path1),nl,
write(' '),
write(Policy1),
write(' Targets: '),
write_targets(Target1),nl,
write(' '),
write(Policy2),
write(' Path = []'),
write_path(Path2),nl,
write(' '),
write(Policy2),
write(' Targets: '),
write_targets(Target2),nl,
write(' '),
write('Target Conflicts: '),
print_list(Permit_results),nl,
print_how_resolved(Policy1,Value1,Policy2,Value2),
print_resolved_conflict_list(Tail).

print_resolved_conflict_list([[Policy1,_,_,Policy2,_,_,_] | Tail]) :-
    policy_owner(Policy1,Owner1),
    policy_owner(Policy2,Owner2),
    user(Owner1,Value1),
    user(Owner2,Value2),
    Value1 = Value2,
    print_resolved_conflict_list(Tail).

%=====
% Print nicely the how the a policy conflict was resolved
%
print_how_resolved(Policy1,Value1,Policy2,Value2):-
    Value1 < Value2,
    write(' '),
    write('Resolved: '),
    write(Policy1),
    write('(Priority = '),
    write(Value1),
    write(')'),
    write(' overrides=> '),
    write(Policy2),
    write('(Priority = '),
    write(Value2),
    write(') '),nl.

print_how_resolved(Policy1,Value1,Policy2,Value2):-
    Value2 < Value1,
    write(' '),

```

```

        write('Resolved: '),
        write(Policy2),
        write('(Priority = '),
        write(Value2),
        write(')'),
        write(' overrides=> '),
        write(Policy1),
        write('(Priority = '),
        write(Value1),
        write(') '),nl.

%=====
% Print nicely a list of elements
%
print_list([]).

print_list([[C|V] | Tail):-
    not(empty(Tail)),
    write(C),
    write(' = '),
    write(V),
    write(', '),
    print_list(Tail).

print_list([[C|V] | Tail):-
    empty(Tail),
    write(C),
    write(' = '),
    write(V).

%=====
% write_path:
% is a helper function used to output
% a list with all the atoms quoted.

write_path([]):- write(']').

write_path([X|[]]):-
    write(X),
    write(']'),!,true.

write_path([X|Tail]):-
    write(X),
    write(','),
    write_path(Tail).

%=====

```

```

% write nicely the target list of a policy
%
write_targets([]):- write('').

write_targets([A,C,_,V|[]]):-
    write(A),
    write(' '),
    write(C),
    write('='),
    write(V),
    write(''),!,true.

write_targets([A,C,_,V|Tail]):-
    write(A),
    write(' '),
    write(C),
    write('='),
    write(V),
    write(' '),
    write_targets(Tail).

%=====
% Print out any message conflicts between a path
% and the links used to compose it.
% First step is to generate the list of conflicts
%     step 2 is to output a list if not empty
%
print_message_conflict_list:-
    not(setof(Link_Message, print_message_conflict_list_helper(Link_Message),_)),fail.

print_message_conflict_list:-
    setof(Link_Message,
print_message_conflict_list_helper(Link_Message),Message_Conflicts),
    write_message_conflicts(Message_Conflicts),fail.

%=====
% Return a list of links and nodes that do not support
% the "messages" require by a policy path
%
print_message_conflict_list_helper(Message_Conflicts):-
    path(Policy,_,_),
    setof(Link_Message, message_conflict(Policy,Link_Message),Message_Conflicts).

%=====
% Print nicely all the policies that contain message conflicts
%
write_message_conflicts([]).

```

```

write_message_conflicts([Message_Conflict|Tail]):-
    write_message_conflict(Message_Conflict),nl,
    write_message_conflicts(Tail).

%=====
% Write out a individual message conflict for a policy
%
write_message_conflict([]).

write_message_conflict([[Policy, Message_list] | Policy_Message_Tail]) :-
    write(Policy),write(' requires message support on the following links'),nl,
    write_message_conflict_helper1(Message_list),
    write_message_conflict(Policy_Message_Tail).

%=====
% Print each link that does not support the "message" on the path
%
write_message_conflict_helper1([]).

write_message_conflict_helper1([[Link,Message] |Tail]):-
    write(' '),
    write('link '),write(Link),write('" requires support for message
"),write(Message),write('"'),nl,
    write_message_conflict_helper1(Tail).

```


THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra, "Routing Policy Specification Language (RPSL)," Internet Engineering Task Force Internet Draft draft-ietf-rps-rpsl-v2-03.txt, April 6, 1999.
- [2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The KeyNote Trust-Management System Version 2," Internet Engineering Task Force: Network Working Group Request for Comments: 2704, September 1999.
- [3] J. Boyle, R. Cohen, D. Durham, S. Herzog, R. Raja, and A. Sastry, "The COPS (Common Open Policy Service) Protocol", Internet Engineering Task Force, Internet Draft draft-ietf-rap-cops-05.txt, December 1998.
- [4] N. Brownlee, "SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups," Internet Engineering Task Force, Internet Draft draft-ietf-rtfm-ruleset-language-07.txt, August 1999.
- [5] J. Case, M. Fedor, M. Schoffstall, J. Davin, "A Simple Network Management Protocol (SNMP)," Internet Engineering Task Force: Network Working Group Request for Comments: 1157, May 1990.
- [6] L. Cholvy and F. Cuppens, "Analyzing consistency of security policies," presented at 1997 IEEE Symposium on Security and Privacy, 1997.
- [7] D. Clark, "Policy Routing in Internet Protocols," Internet Engineering Task Force: Network Working Group Request for Comments: 1102, May 1989.
- [8] A. Guillen, R. N. Kia, and B. Sales, "An architecture for virtual circuit/QoS routing," presented at 1993 International Conference on Network Protocols, 1993.
- [9] J. Honig, D. Katz, M. Mathis, Y. Rekhter, and J. Yu, "Application of the Border Gateway Protocol in the Internet," Internet Engineering Task Force: Network Working Group Request for Comments: 1164, June 1990.
- [10] C. Kunzinger, "Protocol for the Exchange of Inter-Domain routing Information among Intermediate Systems to Support Forwarding of ISO 8473," Internet Engineering Task Force: working draft ISO 10747, April 1994.
- [11] J. Kurose and K. Ross, Computer Networking A Top-Down Approach Featuring the Internet, Addison-Wesley, 2000, pp. 152-153.
- [12] B. Leiner, "Policy Issues in Interconnecting Networks," Internet Engineering Task Force: Network Working Group Request for Comments: 1124, September 1989.

- [13] K. Lougheed and Y. Rekhter, "A Border Gateway Protocol (BGP)," Internet Engineering Task Force: Network Working Group Request for Comments: 1105, May 1989.
- [14] K. Lougheed and Y. Rekhter, "A Border Gateway Protocol (BGP)," Internet Engineering Task Force: Network Working Group Request for Comments: 1163, June 1990.
- [15] K. Lougheed and Y. Rekhter, "A Border Gateway Protocol 3 (BGP-3)," Internet Engineering Task Force: Network Working Group Request for Comments: 1267, October 1991.
- [16] Y. Rekhter, T. Watson and P. Gross, "Application of the Border Gateway Protocol in the Internet," Internet Engineering Task Force: Network Working Group Request for Comments: 1268, October 1991.
- [17] E. Lupu and M. Sloman, "Conflicts in Policy-based Distributed Systems Management," To appear in IEEE Transactions on Software Engineering - Special Issue on Inconsistency, 1999.
- [18] H. Mahon, "Requirements for a Policy Management System," Internet Engineering Task Force, Internet Draft draft-ietf-policy-req-00.txt, September 1999.
- [19] D. Meyer, J. Schmitz, C. Orange, M. Prior, and C. Alaettinoglu, "Using RPSL in Practice," Internet Engineering Task Force: Network Working Group Request for Comments: 2650, August 1999.
- [20] J. B. Michael, E. H. Sibley, R. Baum, and F. Li, "On the Axiomatization of Security Policy: Some Tentative Observations About Logic Representation," presented at Database Security, VI: Status and Prospects, 1992.
- [21] J. B. Michael, "A Formal Process for Testing the Consistency of Composed Security Policies," in Department of Information and Software Systems Engineering. Fairfax: George Mason University, 1993.
- [22] J. Moffett and M. Sloman, "Policy Hierarchies for Distributed Systems Management," IEEE Journal on Selected Areas in Communications, vol. 11, pp. 1404-1414, 1993.
- [23] M. Nossik, F. Welfeld, and M. Richardson, "PAX PDL – a non-procedural packet description language," <http://www.solidum.com/papers/pax-pdel/pax-pdl-00.html>, September 30, 1998.
- [24] The PAIR Project: Policy Analysis of Internet Routing, <http://www.rsng.net/pair/>, 1999
- [25] R. Rajan, S. Kamat, J. C. Martin, M. See, R. Chaudhury, D. Verma, G. Powers, and R. Yavatkar, "Policy Action Classes for Differentiated Services and Integrated Services," Internet Engineering Task Force, Internet Draft draft-rajana-policy-qoschema-01.txt, 5 April 1999.

- [26] M. Steenstrup, "IDPR: An Approach to Policy Routing in Large Diverse Internetworks", *Journal of High Speed Networks*, 1994, pp. 81-105.
- [27] M. Steenstrup, "An Architecture for Inter-Domain Policy Routing", Internet Engineering Task Force: Network Working Group Request for Comments:1478, June 1993.
- [28] M. Steenstrup, "Inter-Domain Policy Routing Protocol Specification: Version 1", Internet Engineering Task Force: Network Working Group Request for Comments:1479, July 1993.
- [29] J. Strassner and E. Ellesson, "Terminology for describing network policy and services," Internet Engineering Task Force, Internet Draft draft-strasner-policy-terms-01.txt, 1998.
- [30] J. Strassner, and E. Ellesson, "Terminology for describing middleware for network policy and services," Internet Engineering Task Force Internet Draft draft-aiken-middleware-reqndef-00.txt, April 30, 1999.
- [31] J. Strassner and S. Schleimer, "Policy Framework Definition Language," Internet Engineering Task Force, Internet Draft draft-ietf-policy-framework-pfdl-00.txt, 17 November 1998.
- [32] J. Strassner, E. Ellesson, and B. Moore. (editor), "Policy Framework Core Information Model," Internet Engineering Task Force: Network Working Group, Internet Draft draft-ietf-policy-core-schema-02.txt, February 1999.
- [33] G. Stone, "Path-based Policy Language", Naval Postgraduate School, Monterey, CA, Manuscript in preparation, August 2000.
- [34] S. Thomas, *IPng and the TCP/IP Protocols*, Wiley Computer Publishing, 1996, pp. 319-350.
- [35] C. Villamizar, C. Alaettinoglu, and D. Meyer, "Routing Policy System Replication," Internet Engineering Task Force, Internet Draft draft-ietf-rps-dist-04.txt, September 28, 1999.
- [36] X. Xiao, A. Hanan, B. Bailey, and L.Ni, "Traffic Engineering with MPLS in the Internet," *IEEE Network*, Vol. 14 No. 2, pp 28-33, March/April 2000.
- [37] X. Xiao and L. Ni, "Internet QoS: A Big Picture," *IEEE Network*, Vol. 13 No. 2, pp. 8-18, 1999.
- [38] G. G. Xie, D. Hensgen, T. Kidd, and J. Yarger, "SAAM: An integrated network architecture for integrated services," presented at Proceedings of 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA, 1998.
- [39] R. Yavatkar, D. Pendarakis, and R. Guerin, "A Framework for Policy-based Admission Control," Internet Engineering Task Force, Internet Draft draft-ietf-rap-framework-01.txt, November 1998.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, Virginia 22060-6218
2. Dudley Knox Library.....2
Naval Postgraduate School
Monterey, CA 93943-5101
3. Chairman, Code CS1
Naval Postgraduate School
Monterey, CA 93943
4. Professor Bert Lundy, Code CS1
Naval Postgraduate School
Monterey, CA 93943
5. Professor Geoff Xie, Code CS1
Naval Postgraduate School
Monterey, CA 93943
6. Professor Bret Michael, Code CS1
Naval Postgraduate School
Monterey, CA 93943
7. Professor John McEachen, Code EC1
Naval Postgraduate School
Monterey, CA 93943
8. Professor Murali Tummala, Code EC1
Naval Postgraduate School
Monterey, CA 93943
9. Communications and Information Systems Department1
Space and Naval Warfare Systems Center
Attn: Mr. Mike Harrison
53560 Hull Street
San Diego, CA 92152-5001

10. Computational Sciences Division1
NASA Ames Research Center
Attn: Marjori Johnson, Senior Scientist
MS 269-2 Moffett Field, CA 94035-1000
11. DARPA / ITO1
Attn: Dr. Mari Maeda
3701 Fairfax Drive
Arlington, VA 22203-1714
12. Directory, National Security Agency.....1
Attn: S353 / Ms. Jan Huff
Fort George G. Meade, 20755-6000
13. Gary Stone2
1026 Upper Mountain Road
Lewiston, NY 14092